2003-09

Software defined radio datalink implementation using PC-type computers

Zafeiropoulos, Georgios

Monterey, California. Naval Postgraduate School

NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

SOFTWARE DEFINED RADIO DATALINK IMPLEMENTATION USING PC-TYPE COMPUTERS

by

Georgios Zafeiropoulos

September 2003

Thesis Advisor:	Jovan Lebaric
Second Reader:	Curtis Schleher

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Software Defined Radio Datalink Implementation Using PC-Type Computers		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Captain Georgios Zafeiropoulos			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT ***(maximum 200 words)***

The objective of this thesis was to examine the feasibility of implementation and the performance of a Software Defined Radio datalink, using a common PC type host computer and a high level programming language. Dedicated transceivers were used, plugged on the PCI bus of host PCs running Windows 2000. Most of the functionality was programmed using the Microsoft Visual C++ language. The tasks to be performed included the channels configuration (number of active channels, center frequencies, sampling and data rates, choice of the appropriate up and down conversion filters), the management of the data transfer between the host computer and the transceiver, the baseband data modulation and demodulation, and the data organization into packets with appropriate headers in order to achieve phase and time synchronization solely by software. A part of the transceivers' configuration was achieved using a configuration utility running in Excel, provided by the manufacturer. Several combinations of M-PSK modulation schemes, channel numbers and datarates were tested in order to measure the performance limits of the system and its ability to perform the required tasks in real-time. The received data streams were further analyzed with the use of Matlab, in order to verify the proper functionality of the communication scheme.

14. SUBJECT TERMS Software Defined Radio, Communications, Datalink, WaveRunner			**15. NUMBER OF PAGES** 190
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SOFTWARE DEFINED RADIO DATALINK IMPLEMENTATION USING PC-TYPE COMPUTERS

Georgios Zafeiropoulos
Captain, Hellenic Air force
B.S., Hellenic Air Force Academy, 1990

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
AND MASTER OF SCIENCE IN SYSTEMS ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2003

Author: Georgios Zafeiropoulos

Approved by: Jovan Lebaric
Thesis Advisor

Curtis Schleher
Co-Advisor

John Powers
Chairman, Department of Electrical Engineering

Dan Boger
Chairman, Department of Information Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The objective of this thesis was to examine the feasibility of implementation and the performance of a Software Defined Radio datalink, using a common PC type host computer and a high level programming language. Dedicated transceivers were used, plugged on the PCI bus of host PCs running Windows 2000. Most of the functionality was programmed using the Microsoft Visual C++ language. The tasks to be performed included the channels configuration (number of active channels, center frequencies, sampling and data rates, choice of the appropriate up and down conversion filters), the management of the data transfer between the host computer and the transceiver, the baseband data modulation and demodulation, and the data organization into packets with appropriate headers in order to achieve phase and time synchronization solely by software. A part of the transceivers' configuration was achieved using a configuration utility running in Excel, provided by the manufacturer. Several combinations of M-PSK modulation schemes, channel numbers and datarates were tested in order to measure the performance limits of the system and its ability to perform the required tasks in real-time. The received data streams were further analyzed with the use of Matlab, in order to verify the proper functionality of the communication scheme.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

The preparation of the thesis is probably the most important task of the student during his two years stay at the Naval Postgraduate School. It is this task which demands from him to combine several of his skills and the knowledge he has acquired by his classes at the school. It also tests his research and development abilities since many times the field he is searching has not been previously developed by anyone else and he is called to proceed one step ahead of wherever everybody else has gone up to now. This is a task which may be quite difficult sometimes and require tremendous effort.

The field of communications has always fascinated me. From the first moment I engaged myself with this subject, I felt really excited. This theme required from me to combine the previous knowledge I had acquired in the fields of communications, digital signal processing and microprocessor programming. It was also a good opportunity to test and improve my programming skills.

Despite the numerous hours of frustration in the lab, when nothing seemed to be working properly due to a wrong bit written in a wrong register, when finally things were tuned and the spectrum analyzer showed the expected signal waveforms, the satisfaction I felt was indescribable. In fact I feel a little sorry now that this project has almost come to an end and I have to separate from my bench in a few days. But, that is life!

I think that the first persons that I must thank are my parents. My current level of success and welfare greatly depends on the endless years of support that they gave me in my childhood, putting themselves in a second level of importance compared to me, the solid rules they have taught me to live with and their love that has always given me the strength to carry on. I just wish they could be present at my graduation!

Then I would like to thank my professors at the Naval Postgraduate School for their patience to share a bit of their wisdom with us. Undoubtedly, none of the work I performed at this thesis would have become possible, without the solid theoretical background that they provided.

A great portion of my thanks must be directed to my thesis supervisor, Professor Jovan Lebaric for his endless support and the valuable advice that helped me overcome a lot of the seemingly unsolvable problems.

I also need to thank the second advisor of my thesis, Professor Curtis Schleher, who very patiently looked at this document and suggested useful changes.

The Hellenic Air Force also deserves my thanks, because it provided me with a state of welfare here in Monterey, making it possible for me to devote my time undistracted to my studies.

Last but not least, I think that I owe my success largely to my wife Tina, the companion of my life for the last few years, who helped me in numerous instances overcome the moments of frustration I felt during my study here at the NPGS when the problems seemed unsolvable, and uncomplainingly endured my endless hours in my study and the school lab.

I hope that you will find the reading of my thesis interesting and that it will give you a fairly good idea of the work I have performed.

LIST OF ACRONYMS AND/OR ABBREVIATIONS

ADC, A/D	: Analog to Digital Converter
AM	: Amplitude Modulation
AGC	: Automatic Gain Control
ASIC	: Application Specific Integrated Circuit
BPSK	: Binary Phase Shift Keying
CIC	: Cascaded Interpolation Comb filter
DAC, D/A	: Digital to Analog Converter
DDS	: Direct Digital Synthesis
DSP	: Digital Signal Processor
EPROM	: Erasable Programmable Read Only Memory
EEPROM	: Electrically Erasable Programmable Read Only Memory
FCE	: Filter Compute Engine
FFT	: Fast Fourier Transform
FIFO	: First In First Out
FIR	: Finite Impulse Response
FM	: Frequency Modulation
FPGA	: Field Programmable Gate Array
GHz	: Gigahertz
GUI	: Graphical User Interface
IF	: Intermediate Frequency
ISI	: Intersymbol Interference
KHz	: Kilohertz
Ksps	: Kilosamples per second
LNA	: Low Noise Amplifier
LPF	: Low Pass Filter
PCM	: Pulse Coded Modulation

PSK	: Phase Shift Keying
QAM	: Qadrature Amplitude Modulation
QDDC	: Quadrature Digital Down Converter
QDUP	: Quadrature Digital Up Converter
QPSK	: Quadrature Phase Shift Keying
RAM	: Random Access Memory
ROM	: Read Only Memory
LNA	: Low Noise Amplifier
LUT	: Look-Up Table
MHz	: Megahertz
Msps	: Megasamples per second
NCO	: Numerically Controlled Oscillator
OOP	: Object Oriented Programming
PLD	: Programmable Logic Device
RF	: Radio Frequency
Rx	: Reception
SDR	: Software Defined Radio
Tx	: Transmission

I: INTRODUCTION

A. THE NEED FOR SOFTWARE DEFINED RADIO

Since early 1980 an exponential increase of cellular mobile systems has been observed, which has produced, all over the world, the definition of a plethora of analog and digital standards. In the current years the industrial competition between Asia, Europe, and America promises a very difficult path towards the definition of a unique standard for future mobile systems, although market analyses underline the trading benefits of a common worldwide standard.

Existing technologies for voice, video, and data use different packet structures, data types, and signal processing techniques. Integrated services can be obtained with either a single device capable of delivering various services or with a radio that can communicate with devices providing complementary services. The supporting technologies and networks that the radio might have to use can vary with the physical location of the user. To successfully communicate with different systems, the radio has to communicate and decode the signals of devices using different air-interfaces. Furthermore, to manage changes in networking protocols, services, and environments, mobile devices supporting reconfigurable hardware also need to seamlessly support multiple protocols, such as IP (Internet Protocol) and MExE (Mobile Execution Environment). Such radios can be implemented efficiently using software radio architectures in which the radio reconfigures itself based on the system it will be interfacing with, and the functionalities it will be supporting.

Most radio receivers and transmitters today are similar to those used decades ago. They consist of dedicated analog circuits for filtering, tuning and demodulating/modulating a specific type of waveform. To make radio systems more flexible, a software-defined radio is currently being developed for both communication and broadcast applications. A software - defined radio is a device

which accommodates a variety of receiver/transmitter programs all on a single hardware platform. The programs on the receiver side perform band pass filtering, automatic gain control, frequency translation, low-pass filtering, and demodulation of the desired signal, and similarly on a transmitter side. Maximizing the number of functions handled digitally, allows the radio to take advantage of the flexibility of the digital signal processing circuit.

B. DEFINITION - CHARACTERISTICS OF THE SOFTWARE DEFINED RADIO

The term **Software Defined Radio** (SDR) was coined by Joe Mitola in 1991 (Ref. 4) to refer to the class of reprogrammable or reconfigurable radios. In other words, the same piece of hardware can perform different functions at different times. The SDR Forum defines the ultimate software radio (USR) as a radio that accepts fully programmable traffic and control information and supports a broad range of frequencies, air-interfaces, and applications software. The user can switch from one air-interface format to another in milliseconds, use the Global Positioning System (GPS) for location, store money using smart card technology, or watch a local broadcast station or receive a satellite transmission. Although the exact definition of software radio is a bit controversial, however, a good working definition is: *a radio that is substantially defined in software and whose physical layer behavior can be significantly altered through changes to its software*.

A typical software defined radio architecture is shown in Figure 1-1. Although a thorough description of the several modules of the platform will be given later on, for the time being let us emphasize the fact of the early signal digitization just after the RF frontend and its subsequent treatment in the discrete domain.

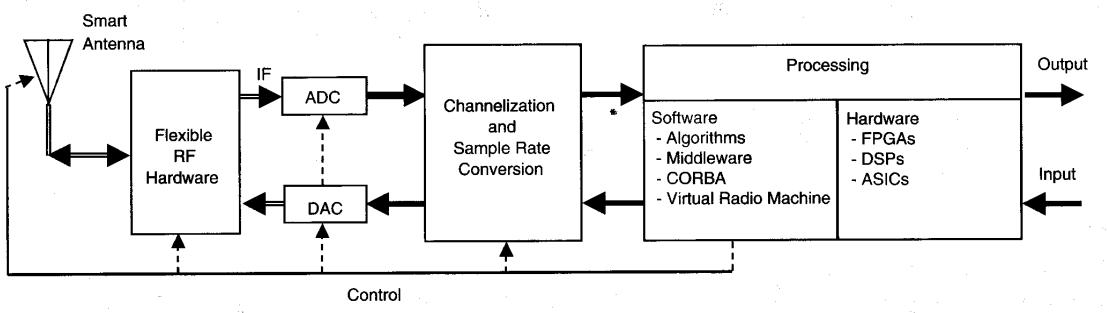


Figure 1-1. A typical Software Defined Radio model (From Ref. 4).

Software radios are emerging in commercial and military infrastructure. This growth is motivated by the numerous advantages of software radios such as:

Ease of design – It is possible to design many different radio products using a common RF front-end with the desired frequency and bandwidth in conjunction with different signal processing software. Thus, it frees the engineer from much of the iteration associated with analog hardware design.

Ease of manufacture – Given the same input in two digital processors and running the same software, they will produce identical outputs. Thus, the move to digital hardware reduces the costs associated with manufacturing and testing the radios.

Multimode operation – A software radio can change modes by simply loading appropriate software into the memory.

Use of advanced signal processing techniques – The availability of high speed signal processing on board the radio allows implementation of new receiver structures and signal processing techniques.

Fewer discrete components – A single high-speed processor may be able to implement many traditional radio functions reducing the number of required components and decreasing the size and cost of radio.

Flexibility to incorporate additional functionality – Software radios may be modified in the field to correct unforeseen problems or upgrade the radio.

The factors that are expected to push a much wider acceptance of software radio are the following five:

Multifunctionality – Software radio reconfiguration capability can support an almost infinite variety of service capabilities in a system.

Global mobility – The ability of the software radio to operate with all the communication standards in different geographical regions of the world.

Compactness and power efficiency – The software radio approach results in a compact and, in some cases, a power-efficient design as the number of systems increases.

Ease of manufacture – In general, digitization of the signal early in the receiver chain can result in a design that incorporates significantly fewer parts, meaning a reduced inventory for the manufacturer.

Ease of upgrades – As new devices are integrated into existing infrastructures, software radio allows the new devices to interface seamlessly, from the air-interface all the way to the application, with the legacy network.

Software radios derive their benefits from their flexibility, complete and easy reconfigurability, and scalability. It is important to ensure that these characteristics are present in the final product. A generic design procedure for software radios follows and demonstrates the interaction between the various

subsystems of the radio design. The following steps focus on the details of these design procedures.

Step 1: Systems engineering – Understanding the constraints and requirements of the communication link and the network protocol allows the allocation of sufficient resources to establish the service given the system's constraints and requirements. In an ideal software radio with the ability to change a number of system parameters in real-time, optimizing an active communications session is a major challenge.

Step 2: RF chain planning – The ideal RF chain for the software radio should incorporate simultaneous flexibility in selection of power gain, bandwidth, center frequency, sensitivity, and dynamic range. However, achieving strict flexibility is impractical and trade-offs must be made.

Step 3: Analog-to-digital and digital-to-analog conversion selection – Analog-to-digital and digital-to-analog conversion for the ideal software radio is difficult to achieve and, in practice, the selection requires trading power consumption, dynamic range, and bandwidth (sample rate). Analog-to-digital conversion selection and vice versa is closely tied to the RF requirements for dynamic range and frequency translation.

Step 4: Software architecture selection - The software architecture is an important consideration to ensure maintainability, expandability, compatibility, and scalability for the software radio. Ideally, the architecture should allow for the hardware independence through the appropriate use of middleware, which serves as an interface between applications-oriented software and the hardware layer. The software needs to be aware of the capabilities of the hardware (both DSP and RF hardware) at both ends of the communications link to ensure compatibility and to make maximum use of the hardware resources.

Step 5: Digital signal processing hardware architecture selection - The core digital signal processing hardware can be implemented through microprocessors, FP-Gas, and/or ASICs. The selection of the core computing elements depends on the algorithms and their computational and throughput requirements. In practice, a software radio will use all three core computing elements, yet the dividing line between the implementation choices for a specific function depends on the particular application being supported.

Step 6: Radio validation - This step is perhaps the most difficult. It is essential to ensure not only that the communicating units operate correctly, but also that a glitch does not cause system-level failures. Interference caused by a software radio mobile unit to adjacent bands is an example of how a software radio could cause a system-level failure, and this is of great concern to government regulators. Given the many variable parameters for the software radio and the desire for an open and varied source of software modules, it is very difficult to ensure a fail-proof system.

C. ABOUT THIS THESIS

The objective of this thesis is to exploit the potential (and the performance) of implementing a software defined radio using standard PC-type computers. Lately, the advances in semiconductor technology have boosted the performance of computers, increasing the processors clock rates to the order of several GHz, while all the critical links needed in order to achieve fast and reliable data transfers (memory, hard disks, I/O buses etc) are now much more optimized and faster than some years ago. Moreover, the internal architecture of the newest generation of processors (like Pentium 4 at 3.06 GHz) make them ideal for multi threaded applications, which is a key fact in designing multi-channel applications for the SDR platform.

The above facts have made us consider quite feasible the implementation of a SDR datalink using dedicated hardware hosted in a commercial PC, with all its functionality programmed in a high level programming language. In order to achieve the above goal, we used two **WaveRunner 253** SDR transceivers, produced by *Red-River Inc*, Richardson TX. The main characteristics of the above cards are:

- Total transmission/reception bandwidth: 3 – 40 MHz.
- Up to 8 fully configurable and programmable transmission/reception channels of up to 3.5 Mbps data rate.
- Possibility to combine individual channels in polyphase filters implementations
- Standard PCI form factor, supporting 32- and 64-bit PCI buses

All the programming was done using Microsoft Visual C++ V.7, which is a part of the *Microsoft Visual Studio NET* programming suite. The channels were configured using a dedicated configuration tool, provided by the manufacturer.

It is the implementation of this effort that this thesis will try to depict. In order to introduce the reader to the theory behind the implementation and make him understand the steps of our effort, the next chapters of this thesis cover the following material:

Chapter II covers briefly all the theory required to understand the functionality of a SDR transceiver. The subjects that are covered include digital signal synthesis, multirate digital signal processing, analog-to-digital and digital-to-analog conversion features and software requirements and specifications. Other subjects such as smart antenna design for SDR, the role of a SDR as an integral part of a radio network and the systems engineering approach to the SDR design, are far beyond the scope of this thesis and will not be covered.

Chapter III presents the architecture of the hardware that was used. The data paths of the transmitter and the receiver are briefly described. More attention is given to the Digital Up Conversion (DUC) and the Digital Down Conversion (DDC) chips which are the “heart” of the devices and their operation ensures the correct functionality of the transceivers. Also, the method of data exchange between the host computer and the card buffer memory is explained. This operation is quite significant, since it was the main focus of our programming efforts described in the next chapter.

Chapter IV is divided in 2 parts: The first part will portray the configuration of the individual channels, using the dedicated configuration tool. The second part explains the structure of our program, the main entities, the methods of achieving specific results on the cards and the methods of interaction between the several threads of the application, depending on the number of active transmission – reception channels. The programming of hardware is a very elaborate process, with hidden dangers in every step of it, which sometimes cannot be easily identified. Also, a thorough knowledge of the hardware functionality is required, since sometimes the most unpredictable things can happen by setting even one inappropriate value in a register.

Chapter V outlines the results of our effort. Although the nature of this thesis is not inherently theoretical, we dare say that according to our knowledge it is the first time that a radio link has been implemented on campus using standard commercial “office” computers and a high level commercial language. So, it will be quite interesting to discover the potential of this system.

The sixth and final chapter presents the conclusions and the areas for further research. The potential and capabilities of the hardware we used are indeed very large. Given the time restrictions of this thesis, only a small part of these capabilities has been exploited. We just proved that this system can be built and that it has an acceptable performance. The real magic of this platform

lies on the flexible and adaptive use of its resources: optimum usage of the available spectrum and automatic reconfigurability are only some of its potentials.

Last but not least, it is the author' s opinion that the control of hardware functionality through software is one of the most interesting things that an electrical engineer can do today. It requires a lot of skills and knowledge in many different areas such as software design, communications and signal processing. It is only the successful "marriage" of these skills that will lead to successful results. The effort of the last months and the pleasure of looking at the results of the several trials have given the author the kind of pleasure that only those who face electrical engineering in general and communications more specifically, not only as a profession, but also as a hobby, can understand.

THIS PAGE INTENTIONALLY LEFT BLANK

II: THEORY REVIEW

A. INTRODUCTION

The architecture of the software radio receiver is quite different from the classical receiver architectures, with the heterodyne receiver being the most dominant one.

The thorough understanding of the principles of operation of a software radio platform requires good knowledge of several technical fields. A representative but not exhausting list of these fields is:

- The conversion of the signal from the analog to the digital domain has moved just before (for the transmitter), or just after (for the receiver) the RF frontend, creating new requirements for faster digital-to-analog or analog-to-digital converters, operating at higher frequencies with acceptable resolution.
- All the treatment of the signal, such as channelization or up and down conversion, is done in the digital domain using the principles of multirate signal processing. However, some or all of the filters used must also satisfy requirements from the communications field (such as the Nyquist property).
- The signals required to feed digital mixers in order to generate useful waveforms are generated entirely in the digital domain as well. There are several methods to do that, each with its advantages and disadvantages.
- Finally, the software that is used to program the functionality of the platform, must posses certain properties in terms of robustness, performance and ability to control the hardware.

The purpose of this chapter is not to analyze exhaustively but rather to highlight the above aspects and provide a brief description of the underlying principles.

B. RADIO RECEIVER TOPOLOGIES

The **Tuned Radio Frequency** (TRF) receiver, shown in Figure 2-1, consists of an antenna connected to an RF bandpass filter (BPF). The BPF selects the signal and the low-noise amplifier (LNA) with the automatic gain control (AGC) raises the signal level for compatibility with the analog-to-digital converter (ADC). This BPF bandwidth relative to the carrier frequency can be quite narrow, while in absolute bandwidth, it may be quite broad.

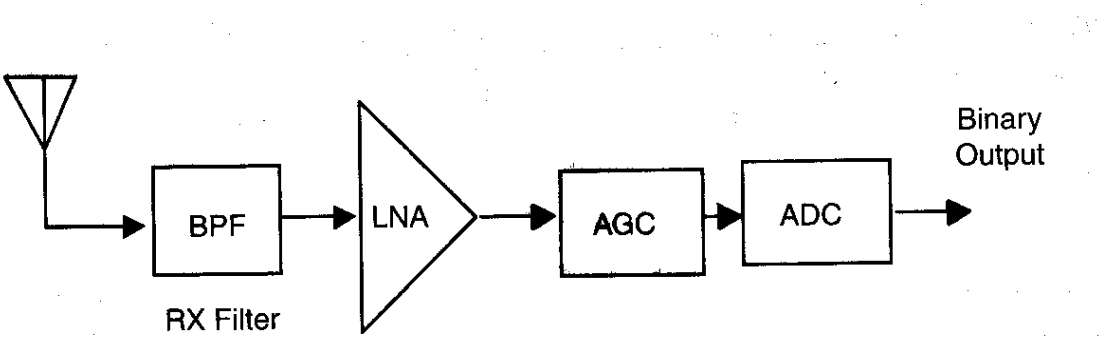


Figure 2-1. TRF Digital Signal Processing Receiver (From Ref. 4).

The primary difficulty in creating a practical TRF receiver is the limitation of the ADC, which must handle high-frequency signals. In addition, given the bandwidth and roll-off limitations of the RF filter, the sampling rate of the ADC must be very high to avoid significant aliasing. High power consumption is inevitable with high sampling rate conversion. Achieving this sampling characteristic is difficult, expensive, and power-intensive, and extreme demands are made of the tunable RF filter to remove interference signals that consume the dynamic range of the ADC. Non-idealities of the ADC, such as jitter and finite aperture size, lead to distortion of the signal. The AGC must adjust its gain to accommodate varying signal levels to utilize the full range of the ADC without

overloading it. However, the especially high gain required for a single-stage AGC in this application may be difficult to control. Nevertheless, the advantage of this approach is the minimal number of analog parts required.

A very popular topology for low-power applications is the single conversion receiver (also known as **homodyne, direct conversion, or zero IF** receiver). This receiver architecture is shown in Figure 2-2. After signal filtering, amplification and gain control, a single mixing stage converts the signal to baseband or near baseband coherently (Figure 2-2a) or incoherently (Figure 2-2b).

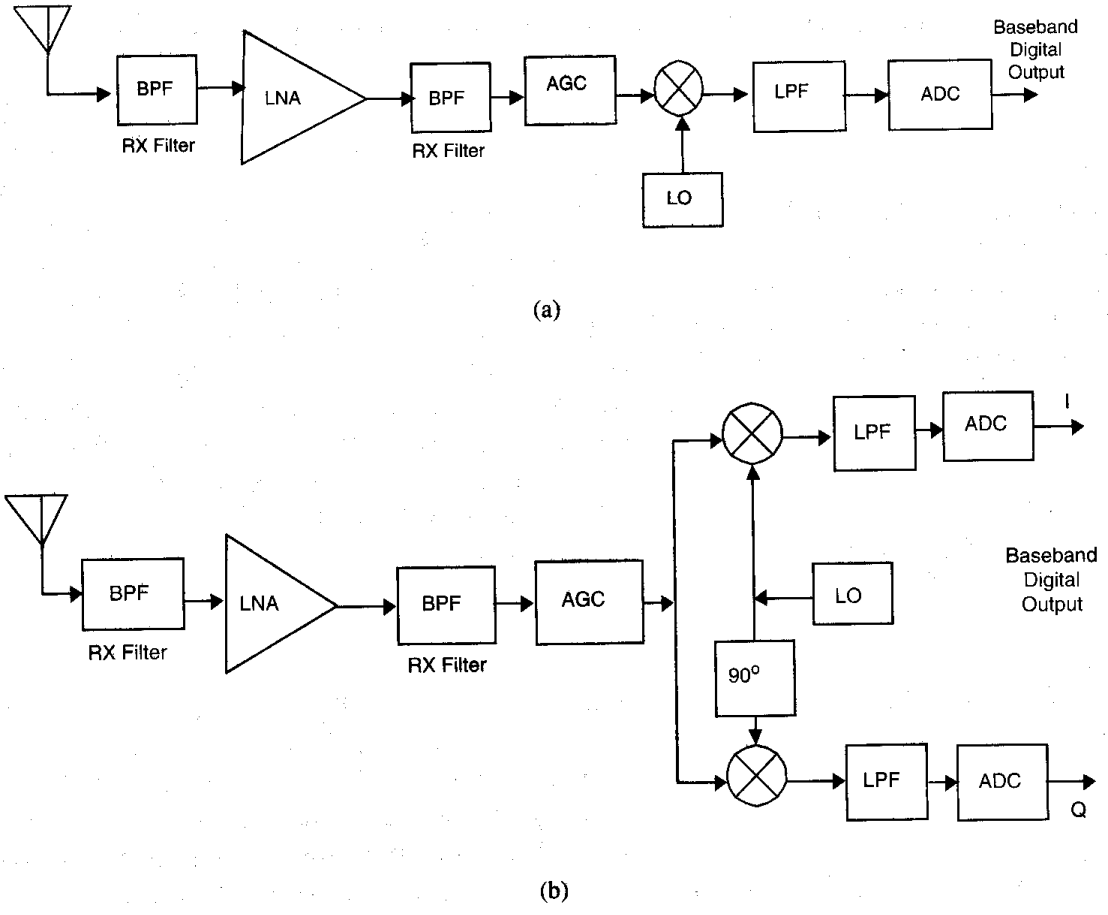


Figure 2-2. Single Conversion Homodyne Receiver for (a) coherent and (b) non coherent reception (From Ref. 4).

In the case of a phase or frequency modulated signal, I and Q downconversion is required since the upper and lower sidebands of these signals contain different information and the sidebands would overlap for a real downconversion. Mixers tend to have high power consumption and, since only one mixer stage (possibly I and Q) is used in the single conversion receiver, the receiver potentially offers good power consumption characteristics. Typically, improved power consumption at the mixer can be traded for dynamic range.

In some cases, rather than directly downconverting the signal to baseband, it may be more convenient to downconvert to some low intermediate frequency at which the signal may be digitized and downconverted by subsequent digital signal processing operations. A more complex LPF with better roll-off characteristics can help reduce out-of-band interference and thus lessen the dynamic range requirement of the ADC, but it could also allow more noise to enter the system (less sensitivity), resulting in non-linear distortion products from the filter.

The most common RF front-end for radios is the **heterodyne** receiver. This receiver, shown in Figure 2-3, is commonly used in analog radios. A heterodyne receiver works by frequency translating the incoming signal to an IF that is fixed and independent of the desired signal's center frequency. When this IF frequency is lower than the center frequency of the received signal's carrier frequency and higher than the bandwidth of the desired signal, the receiver is called a **superheterodyne** receiver. The desired signal is now frequency-translated to a fixed IF can be more easily filtered, amplified, and demodulated. Plenty of good quality RF components are available for standard IF frequencies. Often a superheterodyne receiver involves using two stages of downconversion. Such a dual-conversion receiver has the advantage of relaxed filtering requirements. Because the filtering occurs in stages, the filtering requirements at each stage can be more relaxed than in a single-conversion receiver. That is, by lowering the center frequency of the signal using the first stage of

downconversion, the filter quality factor can also be relaxed because the ratio of center frequency to filter bandwidth is reduced.

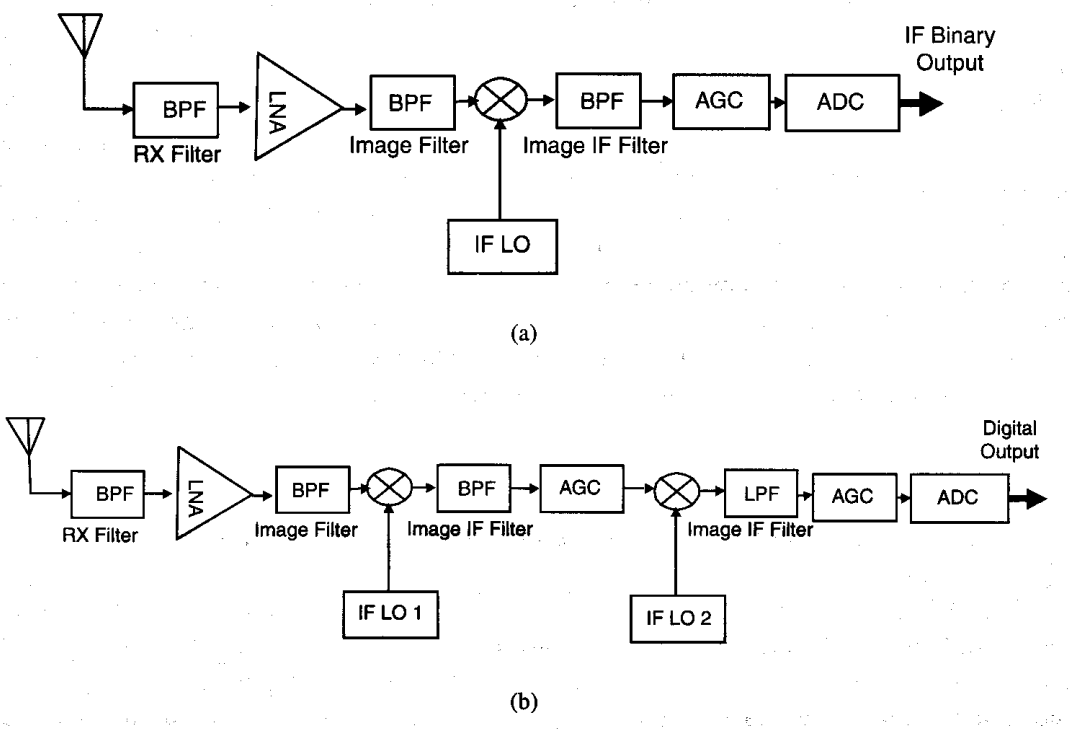


Figure 2-3. (a) Heterodyne Receiver. (b) Dual conversion Superheterodyne Receiver. (From Ref. 4).

At each mixer stage, not only is the signal downconverted, but also a portion of the band at ω_I, the image frequency, is upconverted, which places it on top of the frequency translated desired signal. This problem is illustrated in Figure 2-4. For instance, a 68-MHz LO (ω_{LO}) will downconvert the desired signal by 68 MHz , but the adjacent band, located 136 MHz below the desire signal, will be upconverted to the same frequency range (ω_{IF}, the intermediate frequency) in which the desired signal now lies. To mitigate this self-induced interference, an image filter precedes the mixer to suppress the low-frequency band that might interfere with the desired signal after the mixing operation. Designing the image filter becomes especially challenging if the band of potential interference is heavily occupied with high-power signals. In general, trade-offs exist in the selection of the IF frequency, the image filter, and the post-mixer filter.

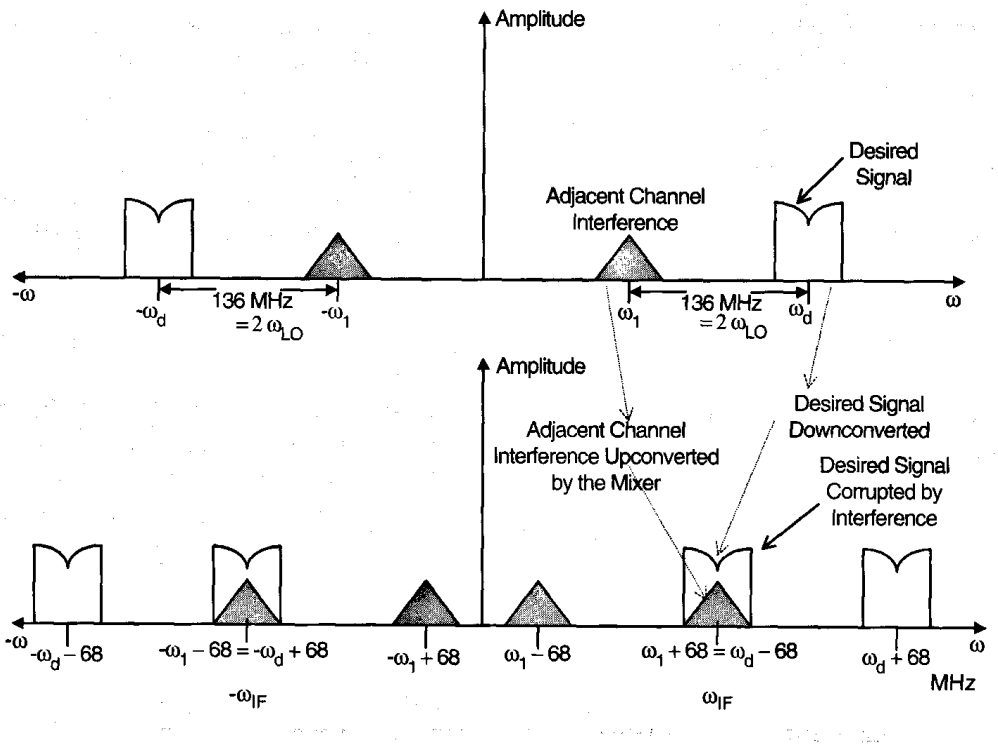


Figure 2-4. The heterodyne receivers image frequency problem (From Ref. 4).

The TRF receiver is better suited for a software radio that supports multiple air-interface modes and multiple bands than the single conversion receiver and particularly better than the heterodyne receiver because the filter requirements for the IF stages make it difficult to support the multiple bandwidths that might be required of a multimode receiver. Retuning a receiver can result in a complex interaction of multiple components comprising the RF chain. The simpler the RF chain, the more predictable its response will be after retuning. The choice of a single or double conversion receiver depends on a number of factors including channel spacing, frequency plan, spurious response, and total gain. In general, the smaller the channel spacing, the more attractive the double conversion receiver becomes because of its ability to narrowly filter the desired signal.

C. MULTIRATE SIGNAL PROCESSING

The conversion of a data stream's sample rate is an important part of digital signal processing. A data stream can be downsampled to a lower sampling rate or upsampled to a higher sampling rate, with the processes known as decimation and interpolation. Often, a non-integer data rate conversion is required and this conversion must be performed in one or multiple stages. These processes are the subject of the following paragraphs.

1. Decimation

Decimation is the process by which high-frequency information is eliminated from a signal to reduce the sampling frequency without resulting in aliasing. A sampled signal repeats its spectrum every 2π radians/sec. If decimation without filtering were performed, aliasing would occur (Figure 2-5).

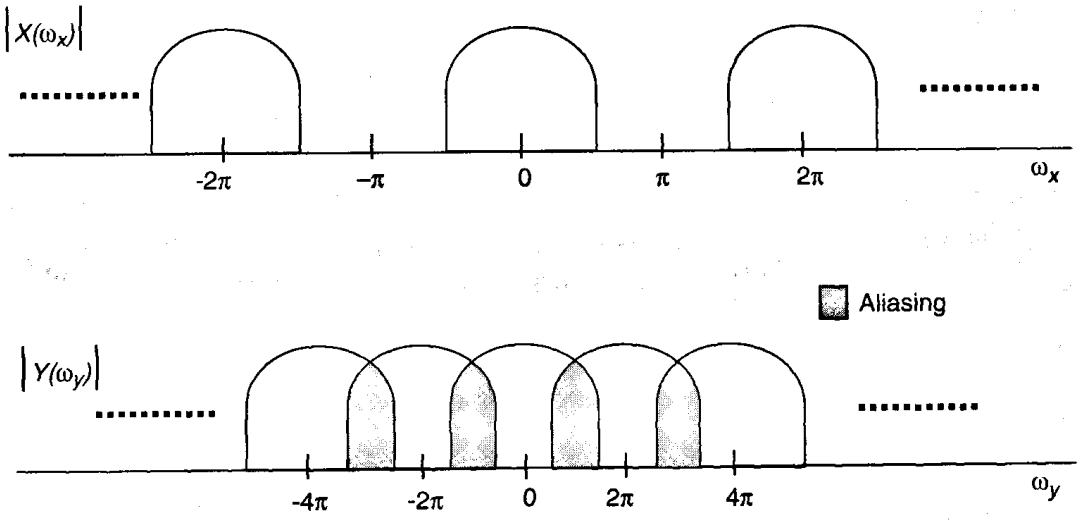


Figure 2-5. Aliased spectrum of an improperly decimated signal. (From Ref. 4).

A block diagram of the decimation process is shown in Figure 2-6, where the operation is composed of lowpass filtering followed by downsampling. The downsampler picks a subset of the samples that are passed through the lowpass

filter (LPF). The LPF used is designed to avoid aliasing and has a cutoff of π/D, the point that allows the non-aliased part of the signal to pass. The end result of the decimation procedure is the content of the original signal below π/D, but it is sampled at a lower rate. Figure 2-7 shows the expected spectrum of the decimated signal.

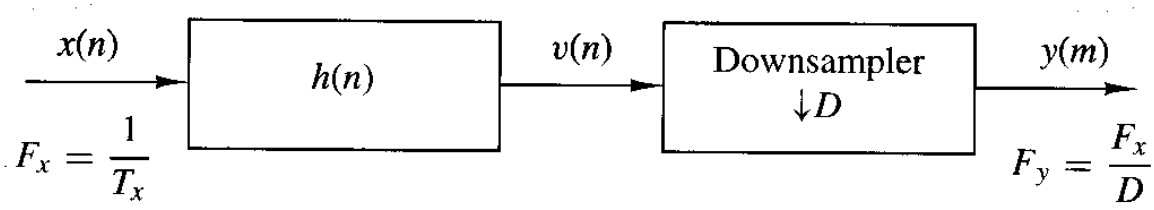


Figure 2-6. Signal decimation by a factor of *D*. (From Ref. 4).

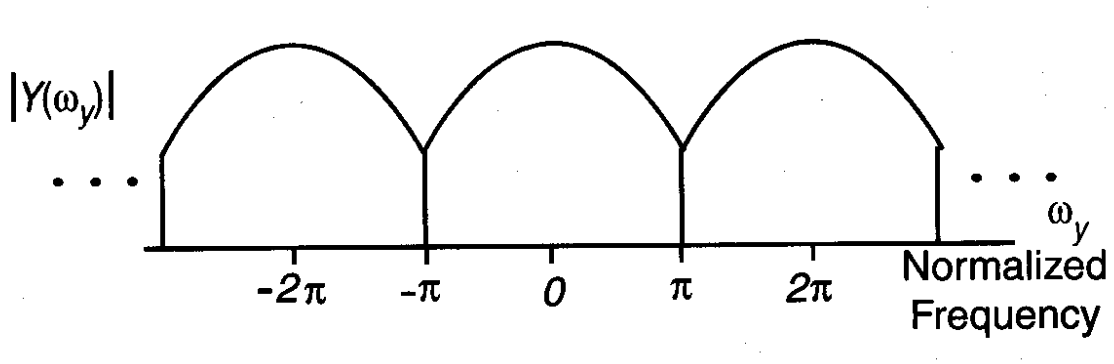


Figure 2-7. Expected spectrum of a decimated signal (From Ref. 4).

The frequency domain representation of the decimated signal is given by the following equation:

$$Y(\omega_y) = \frac{1}{D} X\left(\frac{\omega_y}{D}\right) \qquad |\omega_y| \le \pi . \tag{2.1}$$

Decimation filters out the information in the original signal above π/D (with respect to the original sample rate). A lowpass direct mapping is possible from ω_y to ω_x and vice versa; this relationship is best described as the spectrum spanned by $X(\omega_x), 0 \le \omega_x \le \pi/D$, is also spanned by $Y(\omega_y)$, $0 \le \omega_y \le \pi$.

2. Zero-Insertion Interpolation

Upsampling is the process to increase the number of points per unit time used to describe a signal. When upsampling is employed, no new information is added to the signal. The process of upsampling decreases the time between samples of a signal. This process can be used for matching sampling rates between two systems or as the last step before the digital-to-analog converter (DAC) to help relax the requirements for the reconstruction filter.

In zero – insertion interpolation, zeros are inserted between samples of a signal, generating a new one. This new one, is then lowpass filtered, yielding an upsampled version of the original (Figure 2-8).

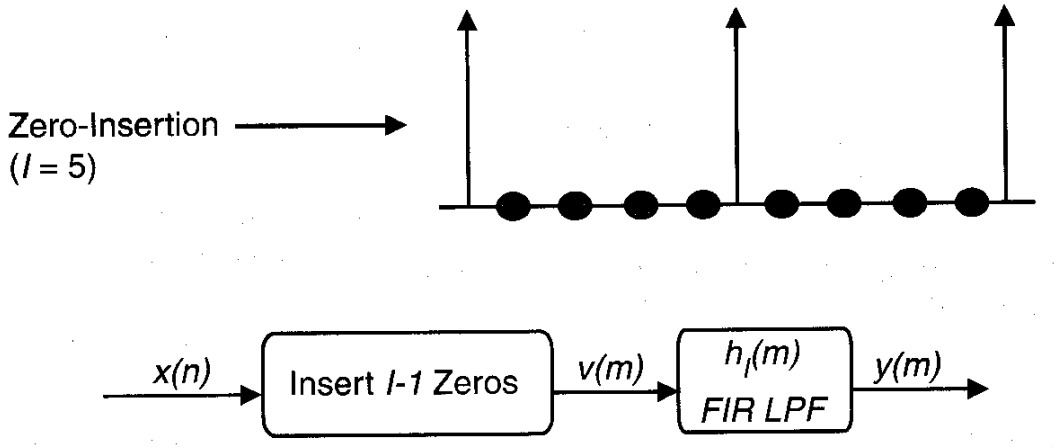


Figure 2-8. Signal interpolation by a factor of 5. (From Ref. 4).

The equation that describes the above procedure is the following:

$$V_{(\omega_y)} = \frac{1}{I} X\left(\omega_y I\right) . \tag{2.2}$$

Note that the $1/I$ factor is included to model the reduction in power (in the normalized scale) resulting from inserting $I-1$ zeros. The above equation shows a contraction of the spectrum; a copy of the spectrum of the original signal is generated every $2\pi/I$ radians/sec instead of every 2π .

3. Zero-Insertion and Raised-Cosine

To minimize inter-symbol interference (ISI), pulse shaping is important. In order to achieve this, a Nyquist filter, such as a raised-cosine pulse shaping filter, can be used. For example, if the upsampling of a pulse code modulation (PCM) signal is to be performed at the transmitter, the upsampling and raised-cosine filtering can be combined to simplify the overall design.

This implementation is performed by using the zero-insertion interpolation method described earlier but with a raised-cosine filter combined with the lowpass interpolation filter as shown in Figure 2-9.

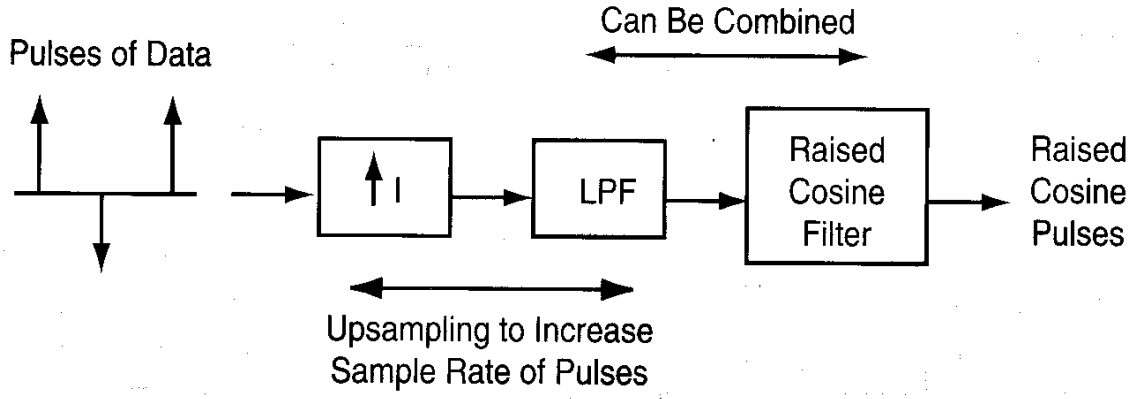


Figure 2-9. Combined upsampling and Raised-Cosine filtering. (From Ref. 4).

4. Non-Integer-Rate Conversion

Non-integer-rate conversions are achievable through the use of cascaded interpolations/decimations such that a total rate change of I/D is achieved.

Figure 2-10 shows a block diagram of the implementation of a non-integer-rate conversion. After interpolation by a factor of *I*, the signal is filtered by a LPF having a cutoff frequency of π/D and subsequently it is decimated by a factor of *D*. So, the overall rate conversion is *I/D*.

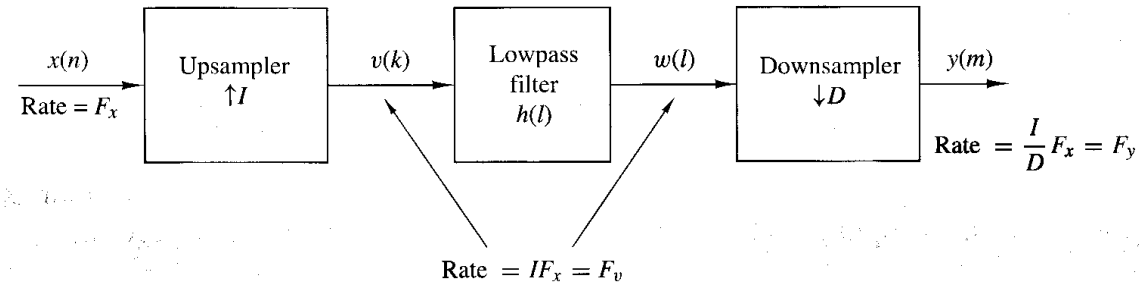


Figure 2-10. Non-integer datarate conversion (From Ref. 3).

The following equation describes the above procedure mathematically:

$$Y(\omega_y) = \left\{ \frac{I}{D} X\left(\frac{I}{D} \omega_y \right) \right\} \qquad |\omega_y| \le \min\left(\frac{\pi}{I}, \frac{\pi}{D} \right) . \tag{2.3}$$

5. Sampling Rate Conversion by Stages

The decimator and interpolator discussed so far are of a single-stage structure. When large changes in sampling rate are required, multiple stages of sample rate conversion are found to be more computationally efficient. Most practical systems employ a multi-stage structure, resulting in a considerable relaxation in the specifications of anti-aliasing (decimation) or anti-imaging (interpolation) filters in each stage compared to a single stage realization.

The decimation in Figure 2-11 can be realized in three stages if the decimation factor D can be expressed as a product of three integers: D_1, D_2 and D_3. Referring to Figure 2-11, in the first stage, the signal $x(n)$ is decimated by a factor of $D_1 = 15$. The output is further decimated by $D_2 = 3$ in the second stage and the output of the second stage is decimated by a factor $D_3 = 2$ in the third stage, resulting in an overall decimation of $x(n)$ by $D = (D_1 D_2 D_3) = 15 \times 3 \times 2 = 90$. The filters $H_1(z)$ and $H_2(z)$ are so designed that the aliasing in the band of interest is below a prescribed level and that the overall passband and

stopband tolerances are met. The filter of the final stage $H_3(z)$ may be quite sharp, but its sampling rate is much lower than the original one, reducing significantly the overall computational burden. This multi-stage sampling rate conversion system requires less computation and offers more flexibility in filter design.

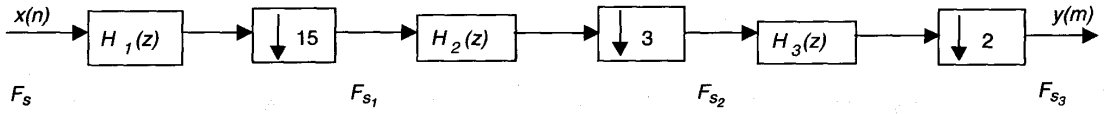


Figure 2-11. Decimation realized in 3 stages (From Ref. 3).

6. Cascaded Integrator Comb Filters

In software radio systems, sample rate changes can be very large, with changes from many tens to MHz to around 100 kHz being common. Of course, such a requirement leads to large order and high-rate digital filters, which can easily become a bottleneck in the overall system design. A cascaded integrator comb (CIC) filter can be used to reduce the computational demands. A CIC filter is what the name indicates: a cascade of simple integrators (accumulators) and a cascade of comb filters (delay and subtract from current sample). The CIC filter can implement an interpolation or decimation filter that uses only delay and add operations and thus is well-suited for FPGA and ASIC implementation. Furthermore, the same basic filter structure can be used to handle variable sample rate conversion.

The CIC implementation of a decimation filter is the cascade of an integrator stage, a decimation procedure, and a comb stage as shown in Figure 2-12. To analyze the CIC filter's response, combining the integrator and comb stages into a single transfer function is important to reduce the complexity of the analysis. However, to reduce the computational expense of the operation, the

implementation of the filter is performed in two separate sections, before and after the decimation.

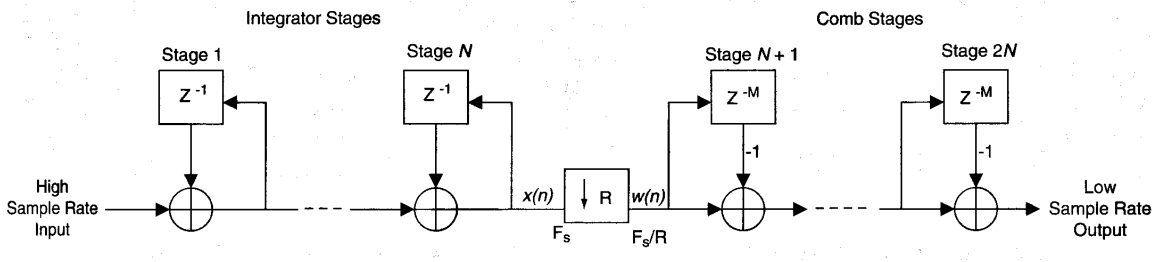


Figure 2-12. A CIC decimation filter (From Ref. 4).

The frequency response (with respect to the higher input sample rate) is:

$$\left|H(\omega)\right| = \left[\frac{\sin\left(\frac{\omega RM}{2}\right)}{\sin\left(\frac{\omega}{2}\right)}\right]^{N}, \tag{2.4}$$

where:

N is the number integrator and comb stages,
M is the differential delay of each comb stage,
R is the decimation rate and
ω is the higher input frequency.

By letting $\omega' = \omega/R$, the frequency response with respect to the decimated sample rate is found to be:

$$\left|H(\omega')\right| = \left|\frac{\sin\left(\frac{\pi M\omega'/2}{2}\right)}{\sin\left(\frac{\pi\omega'}{2R}\right)}\right|^{N}. \tag{2.5}$$

As an example, the frequency response of that equation is plotted in Figure 2-13 for $N = 4$, $M = 1$, $R = 7$ for a cutoff frequency $f_c = 1/8$. The input sample rate is 7 and the output sample rate is 1.

Note that above a normalized frequency of 1, the transfer function will fold and we will have aliasing, but its magnitude will be less than 50 dB down from its maximum value. Moreover, since the CIC filter will most probably be followed by a decimation stage with a FIR filter with a cutoff frequency of π/D (where D is the decimation rate), the aliasing in the useful portion of the spectrum will be even less.

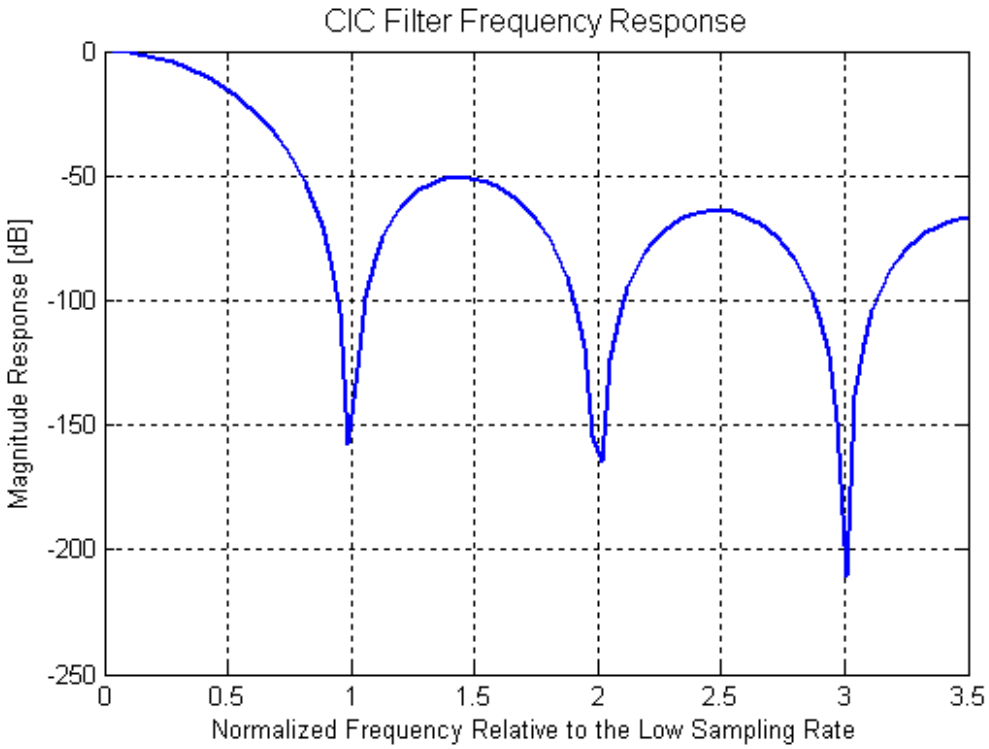


Figure 2-13. Frequency response of a CIC filter.

7. Polyphase Decimation and interpolation

The decimation and interpolation procedures can be also implemented by the polyphase filters (Figures 2-14 and 2-15). In this implementation, the decimation or interpolation procedure is decomposed into a sum of D (or I) parallel filtering stages. In the decimation process the filters $p_k(m)$ are formed by decimating $x(m+k)$ by a factor of D. In the interpolation process, each of the I stages contributes a sample at the output.

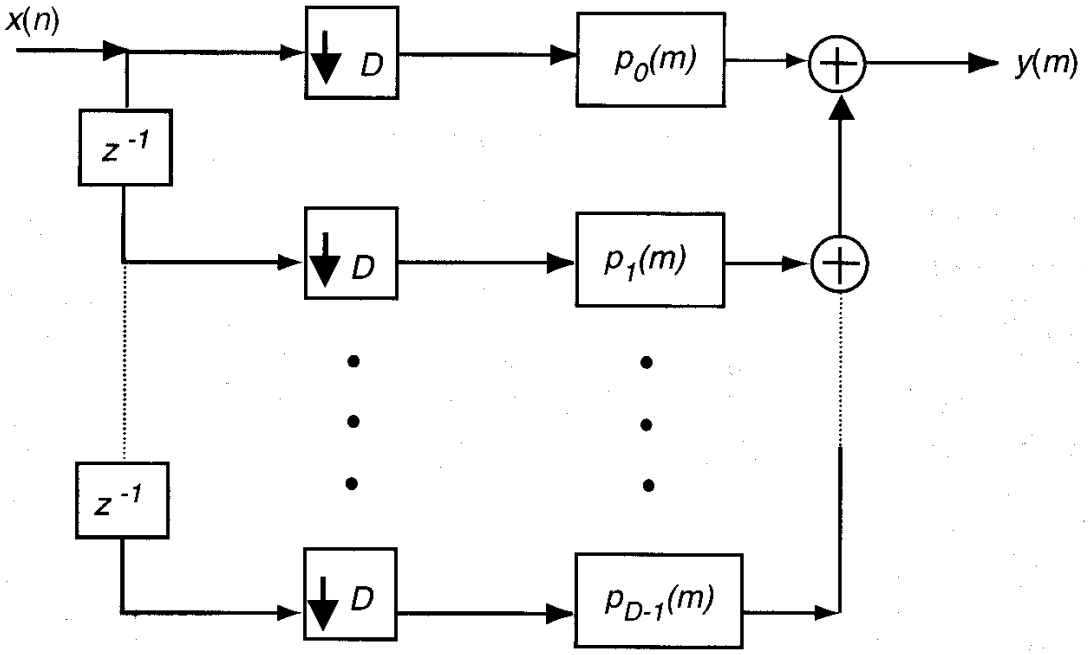


Figure 2.14. General structure of a Polyphase Decimator (From Ref. 4).

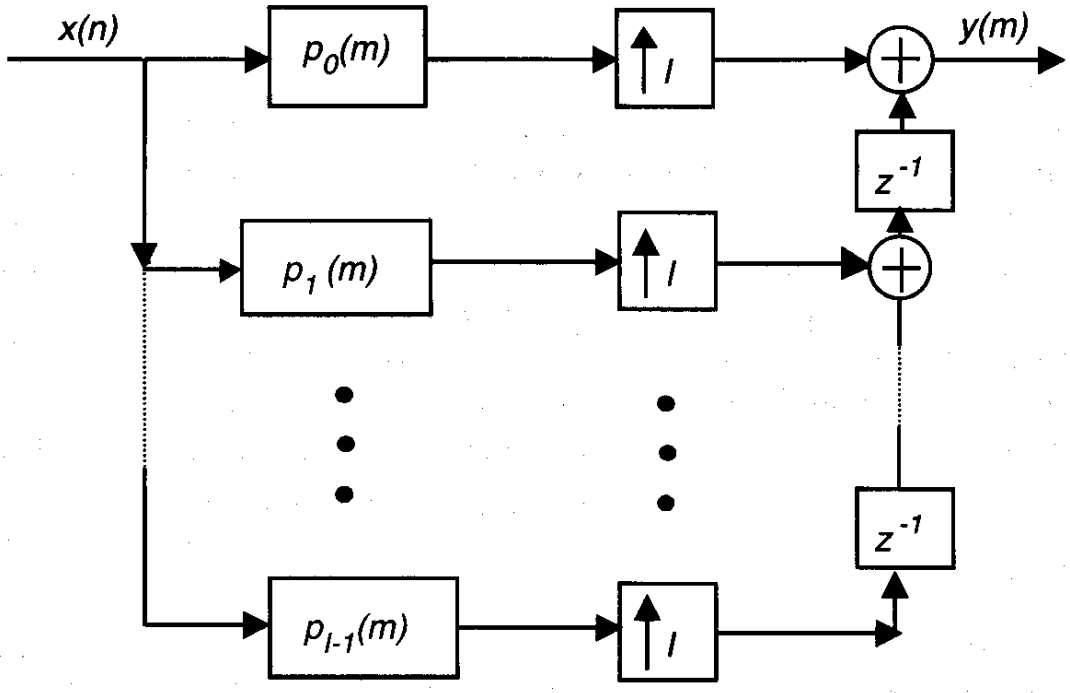


Figure 2.15. General structure of a polyphase interpolator (From Ref. 4).

With the use of the polyphase filters, the filtering occurs always at the lowest frequency, thus reducing the computation burden significantly.

Polyphase implementation of the decimation process of the reception channels constitutes a characteristic of the hardware we are going to use. However, since we will not deal with it extensively, we mention it here just for reference.

D. DIGITAL GENERATION OF SIGNALS

The synthesis of waveforms, especially sinusoidal signals, is an important part of a communication system. Sinusoidal signals are typically used in many of the processing steps, such as modulation, pulse-shaping, and filtering.

Analog techniques have long dominated frequency synthesis. Analog frequency techniques are based on bulky analog devices such as quartz crystals, inductors, capacitors, and mechanical resonators. Digital techniques began to gain prominence in communication systems because of their superior accuracy and immunity to noise and because they are easy to manufacture with very large scale integration (VLSI). Direct digital synthesis (DDS) techniques generate signals directly in discrete time. Any arbitrary waveform can be generated for digital communication systems, as the amplitude, frequency, and phase can be varied to create a modulated signal.

Direct digital synthesizers allow the implementation of digital modulation techniques, after which the signals can be converted to analog signals for transmission. Amplitude modulation (AM) can be created by multiplying the sinusoidal output of the ROM with the modulating signal before passing it through the DAC. Phase modulation (PM) is created by changing the instantaneous phase angle, i.e., by using the modulating signal to alter the input to the ROM (phase). Frequency modulation (FM) is created by varying the instantaneous

frequency, and this is accomplished by using the modulating signal to increment the phase.

1. Comparison of Direct Digital Synthesis with Analog Signal Synthesis

The digital nature of DDS makes it possible to set the frequency of the output wave-form more precisely than analog techniques. In analog systems, the frequency is controlled with analog components, resulting in poor stability due to drift in the components, poor frequency resolution due to limitation in analog dials, and difficulty with digitally controlled tuning. Furthermore, analog signal generators stray in frequency over time. Changes in temperature, humidity, and other variables can affect the output of the analog oscillator. The instrument's overall accuracy varies with time and from one unit to the next. Precise generation of signals leads to the feasibility of very close channel spacing, which is very significant for narrowband modulation formats. Analog systems are generally limited to less demanding applications in which tight frequency control and accuracy are not crucial. Fine frequency steps are achieved easily with a DDS because relatively small increases in circuit complexity can add a decade of additional resolution. Most of today's DDS designs provide step sizes finer than 1 Hz and many can achieve 1 mHz or smaller.

A summary of the advantages and disadvantages of the DDS is presented in the table 2-1.

2. Approaches to Direct Digital Synthesis

There are two basic approaches for generating signals directly from digital hardware. The first is commonly referred to as the ROM Look Up Table approach, which can also be used to generate sinusoidal signals. The sampled values of the sine waveform are stored in ROM and are output periodically

through a DAC to generate the output waveform. The second approach, pulse output DDS, uses a phase accumulator to obtain a series of periodic pulses (or a rectangular or sawtooth waveform) from which other waveforms can be created.

Property	Advantages	Disadvantages
Precision	• Possible to set frequency accurately • Can achieve very high resolutions (<1mHz)	
Flexibility	• Very easy to change the parameters	
Ease of Implementation	• VLSI implementations are inexpensive and readily available	
Switching Frequency	• Possible to have very high switching speeds, within 1μs • Output is smooth and transient free during frequency change • Possible to have continuous phase during frequency switching	
Size of the Equipment	• Can be implemented at a fraction of the size of a similar analog synthesizer	• Techniques for reducing spurious signals, e.g., hybrid DDS-PLLs, can increase the size of the system
Bandwidth	• Can be varied by changing the clock speed • Bandwidth can be increased by using a DDS-driven PLL	• Output frequency limited by the Nyquist frequency to half the DDS clock rate (Fclk/2) • In practice, limited to Fclk/4
Spectral Purity	• Possible to get very high quality of signals when the size of the accumulator is an integral multiple of the step size and there is no phase truncation	• Lots of spurious signals when there is phase truncation or other jitter • Need to use special techniques to reduce spurious responses, e.g., ROM compression, hybrid DDS-PLL, or randomization

Table 2-1. Comparison of Direct Digital Synthesis with Analog Signal Synthesis

3. Pulse Output Direct Digital Synthesis

One of the simplest forms of a DDS system is a pulse output DDS (PO DDS) system. This DDS approach generates pulse, sawtooth, or rectangular waveforms. Other waveforms can be created from these basic waveforms. The

idea is to create the rectangular waveform by cycling through an accumulator as a way to create an adjustable pulse frequency from a stable high-frequency driving clock. PO DDS consists of an N bit adder and register to form an accumulator. A frequency word Δ_r, is added to the accumulator once every clock period, T_{clk} . Figure 2-16 shows a PO DDS.

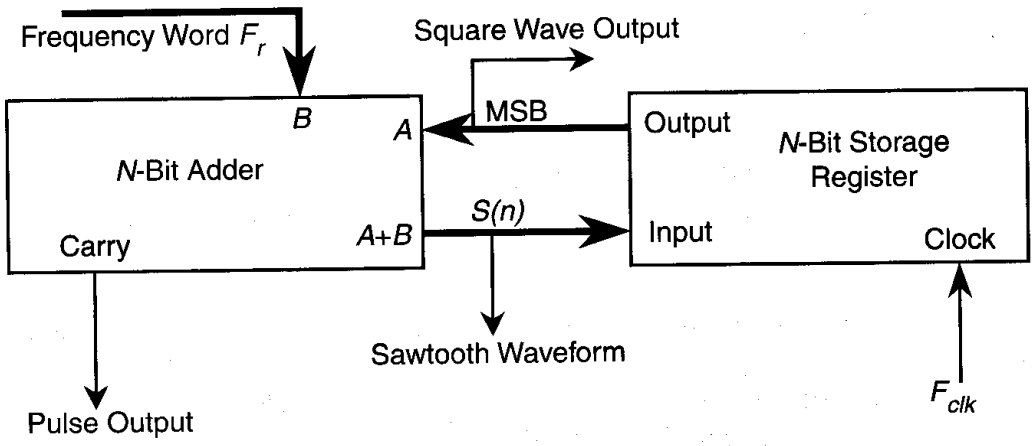


Figure 2-16. Pulse Output Direct Digital Synthesis (From Ref. 4).

The output of the accumulator, $S(n)$ at time n, is given by $S(n) = \left(S(n-1) + \Delta_r\right) \bmod 2^N$ and is performed in modulo 2^N arithmetic. The accumulator will overflow, and the counter resets on average once every $2^N / \Delta_r$ clock periods. The average frequency for which the counter is reset is

$$F_{out} = \frac{\Delta_r}{2^N} F_{clk} . \tag{2.6}$$

The output of this synthesizer could be the carry output of the accumulator for a pulse output or the most significant bit (MSB) of the accumulator to represent the approximate square wave output, or $S(n)$, for a sawtooth waveform.

4. Rom Look-Up Table Approach

The ROM LUT approach uses sampled values of a periodic function stored in a ROM. Every clock cycle, a value of the periodic function stored in the ROM is output through a DAC to generate the synthesized signal. The output from the DAC, however, is a distorted analog signal due to the sample and hold nature of the waveform. Therefore, the signal obtained at the output of the DAC is passed through LPFs and amplifiers to obtain the final analog waveform. Sometimes it is advantageous to perform digital filtering before the digital-to-analog conversion to compensate for the distortion of the non-ideal analog filters.

DDS-based function generators are based on a single crystal oscillator, which generates a reference clock frequency. The structure of a DDS system using a ROM LUT is shown in Figure 2-17. The adder and register function as an accumulator and increment the output value by Δ_r at each clock cycle. For example, if the first adder output is zero and the phase increment Δ_r is 17, then the second output would be 17, the third would be 34, the fourth 51, etc.

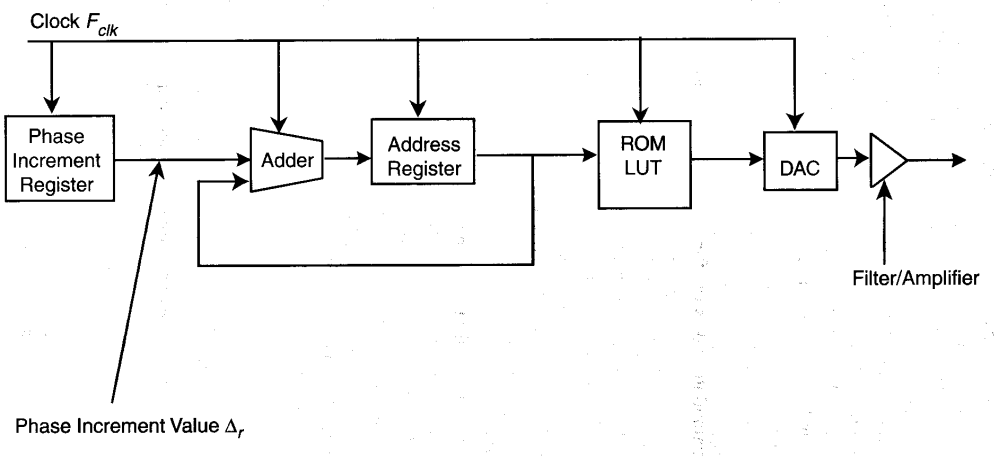


Figure 2-17. ROM LUT Direct Digital Synthesis (From Ref. 4).

The output of the accumulator takes the form of an address used by a ROM LUT that contains the waveform samples. The number of clock cycles needed to step through the entire ROM LUT defines the time period of the

waveform. The waveform period is determined by Δ_r. The LUT holds the digital representation of the desired waveform, which is made up of digital words that define the amplitude of the waveform as a function of phase. The address generated by the adder represents the phase value of the waveform.

The address generator sequentially reads the table of digital values out of the memory and passes them to a DAC to generate the output waveform. The DAC changes each of these digital words into an analog voltage, which is fed through filters to reduce the distortion and amplifiers to produce an analog signal. The period of the output waveform is based on the phase increment value Δ_r and the frequency of the clock signal F_{clk}.

5. Performance Assessment of the DDS Systems

The major drawbacks of a DDS system are spectral purity and sideband noise. The sources of spurious signals are amplitude and phase truncation due to the limited number of bits used for their representation, as well as DAC non-linearities. Several methods are used to mitigate these problems, such as the randomization (Ref. 4). A block diagram of the Wheatley's procedure is shown in Figure 2-18.

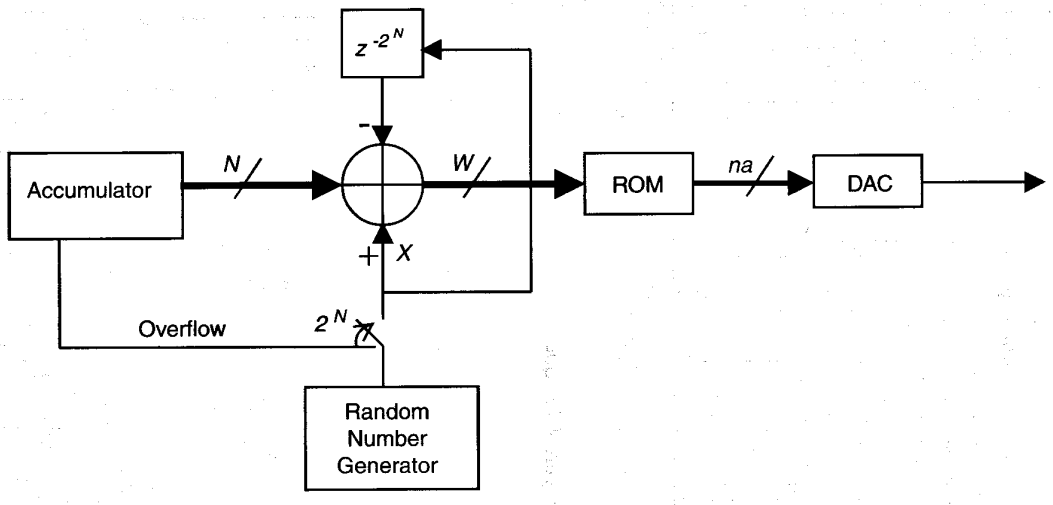


Figure 2-18. Wheatley's procedure (From Ref. 4).

The method consists of adding a sequence of random numbers to the contents of the accumulator in a prescribed manner, in order to convert the discrete harmonic signals into a continuous noise floor, whose level is much lower than that of the harmonic signal. Figure 2-19 shows the spectrum of a DDS, before and after the Whealtey method has been applied.

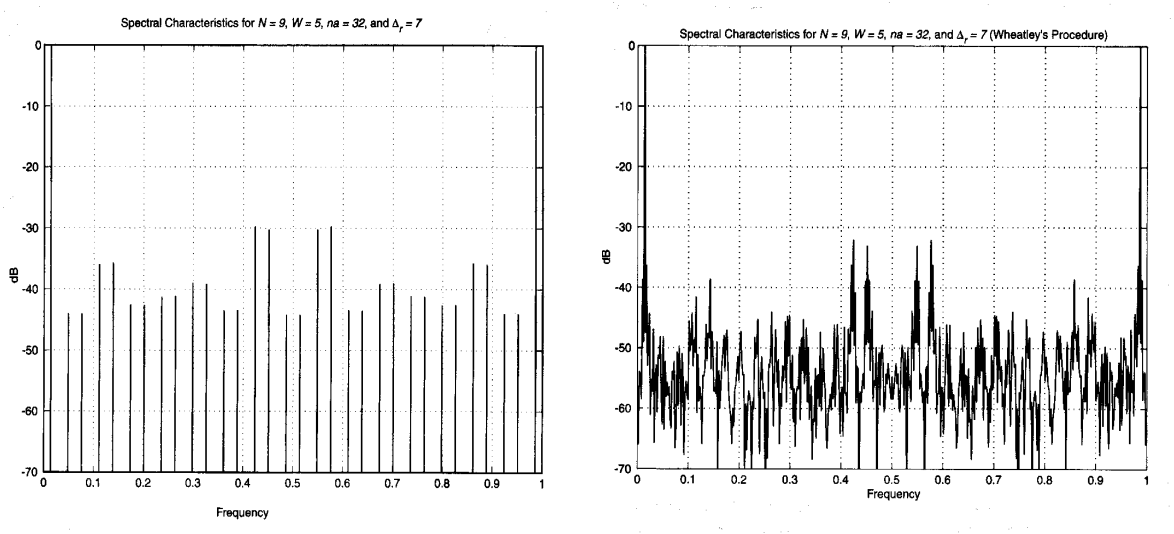


Figure 2-19. DDS spectrum before and after the application of Wheatley's method (From Ref. 4).

E. ANALOG-TO-DIGITAL AND DIGITAL-TO-ANALOG CONVERSION

The digital-to-qnalog converters (DAC) and the analog-to-digital converters (ADC) constitute in many instances the determining factor for the performance of the software radio, since they impact the power consumption, dynamic range, bandwidth and total cost.

Ideally, the data conversion should take place immediately after the antenna in the receiver chain. The RF signal should be sampled and all the downconversion procedure should be carried out entirely in the digital domain.

However, this process would place some extreme constrains on the data converter. It would need:

- Very high sampling rate, in order to support wide signal bandwidths
- A large number of quantization bits, in order to support a high dynamic range.
- A large operating bandwidth (in the order of hundreds of MHz or even some GHz) to accommodate a greatly varying range of signal frequencies.
- A large spurious-free dynamic range to allow the recovery of small-scale signals in the presence of strong interferers.
- Small power consumption, while simultaneously meeting all the above criteria.

Unfortunately, the current capabilities of the available technology do not make it possible to fabricate converters meeting the above specifications. So, an RF frontend is included after the antenna, at which the signal is downconverted to an appropriate IF frequency and data conversion takes place at that frequency

F. CHOICE OF THE APPROPRIATE HARDWARE

The choice of the hardware composition of a software radio is a key step in its design. This choice is dictated by the requirements in four main areas: flexibility, modularity, scalability and performance.

There are three main categories of digital hardware available: Digital Signal Processors (DSP), Field Programmable Gate Arrays (FPGA) and Application Specific Integrated Circuits (ASIC). Each of them exhibits a certain level of reprogrammability, that is, an ability to change the device software.

1. Digital Signal Processors (DSP)

A DSP is designed to support high-performance, repetitive, numerically, intensive tasks and very high I/O performance. DSPs are designed to include special functional units in the hardware as well as special instructions in the microcode. Some DSPs provide for several accesses to memory in a single cycle. Program flow control features speed up the execution of common DSP tasks like FFT or Viterbi decoding. Large accumulators in DSPs help reduce precison problems. Certain DSPs are also optimized for specific applications like wireline communications, wireless communications, or general control applications.

2. Field Programmable Gate Arrays (FPGA)

The FPGA was introduced in the mid-1980s as a device to process digital logic. FPGAs were designed for multilevel circuits, which means they could handle complex circuits on a single chip. Since FPGAs were prefabricated, they were quicker to use and less expensive. The volatility of SRAM makes the FPGA attractive for digital systems. FPGAs are now used in various configurations, as in multimode and reconfigurable systems, and are very useful in meeting the needs of a software radio system.

Like programmable logic devices (PLDs), FPGAs are also completely prefabricated, but they are optimized for multilevel circuits rather than two-level logic and contain special features for customization. These properties allow them to handle much more complex circuits on a single chip but often sacrifice the predictable delays of a PLD. Several kinds of FPGAs exist, such as SRAM cells, erasable programmable read-only memory (EPROM), electrically-erasable programmable read-only memory (EEPROM), or anti-fuses.

3. Implementing DSP Functions in FPGAs

There are two ways of implementing DSP functions on an FPGA, bit serial and bit parallel distributed arithmetic. Both methods are scalable, which allows the designer to optimize the design for performance and density. Bit serial distributed arithmetic is an implementation technique that processes parallel data flow structures bit sequentially, thereby allowing multiple functions to performed simultaneously. For example, when implementing a sixteen–tap FIR filter, all sixteen data samples are multiplied in parallel in a bit serial process. Bit parallel distributed arithmetic is a similar technique but with multiple bits being processed in parallel, allowing the overall performance of the design implementation to scale proportionately to required resources. At high-performance levels, e.g., thirty to seventy million samples per second (MSPS), the FPGAs can be partitioned to perform all operations in parallel, minimizing the number of clock cycles required to perform a function. At lower-performance levels, e.g., one to ten MSPS, bit-sequential operations allow more efficient resource use.

To better understand the performance and cost benefits using FPGA devices, consider the design of an eight-bit, sixteen-tap FIR filter. Most programmable DSPs can perform a memory access control (MAC) function in one clock cycle. Therefore, implementing the FIR filter in a 66-MHz DSP would yield a theoretical maximum saple rate of 66 MHz/16 taps = 4.125 MSPS, excluding memory overhead operations. In contrast with a FPGA-based implementation, the designer could use bit-serial distributed arithmetic where all sixteen taps are processed in parallel. Using the same 66-MHz clock rate, this implementation can process the data at 15 ns per bit; for eight-bit data this equates to 8.33 MSPS, twice the sample rate of 66-MHz programmable DSP device.

4. Using a Combination of DSPs, FPGAs, and ASICs

When system performance demands exceed existing processor capability, a number of approaches are used to solve the problem, including custom integrated circuits (ASICs), function specific DSP cores, multi-processor architectures and reconfigurable architectures. A popular trend is the use of DSPs as cores inside ASICs to provide some degree of flexibility to the ASIC. Another popular method of boosting the performance of a DSP places multiple DSPs in parallel along with a high-speed memory. FPGAs can also be used to enhance a given DSP. The parallel paths of an algorithm can be implemented on the FPGA while the DSP handles the sequential and other general sections of the algorithm. The FPGA can be programmed to perform any number of parallel paths. These operational data paths can consist of any combination of simple and complex functions, such as adders, barrel shifters, counters comparators, correlators and so forth.

G. OBJECT ORIENTED PROGRAMMING (OOP) AND THE SDR

The main principle of the OOP is the existence of objects. Objects are autonomous entities with their own functionality and data, which constitute distinct instantiations of classes. Classes are prototype entities which define what the functionality and data will be. Objects and classes have useful properties such as:

- **Inheritance**: a derived - or child - class can inherit all the attributes and functionality of the parent class and add its own functionality.
- **Polymorphism**: the functionality of the parent class can be modified, or take alternate forms, in order to serve the specific needs of the derived class.
- **Encapsulation**: The access to the functionality and data of the class can be controlled by the class creator. In this way, enough information

can be accessible by the outside programmer in order to take advantage of the class functionality, but not adequate enough to disturb the normal behavior of the class.

- **Overloading**: This feature allows the class to create multiple behaviors according to the situation and thus adapt to a variety of conditions.

A radio platform can be broken up into discrete objects, each of which interacts with other objects in the system. This object oriented way of looking at the world can be translated into software and the several programs that control the functionality of the platform can be based on discrete objects, each of which interacts with other objects and the system through well defined interfaces.

The use of objects will allow the software to mimic, at least in form, the layout of a real system. The object-oriented approach, through the use of abstract classes, can be used to create a framework from which development can be structured and system integration can be simplified.

There are several possible candidate languages which can serve as the development platforms for writing the software. Each one has advantages and disadvantages. In the following two paragraphs, we will examine two of them, C++ and Java. This is due to the wide acceptance of the above languages from the programming community, as well as to the extensive features that they offer.

1. C++

So far, C++ seems as the most suitable language for the development of the SDR applications. This is due to several reasons:

- It is an inherently object oriented programming language.

- Since it is derived from the classical C language, it has been in use for many decades and is widely accepted.
- It offers the possibility for detailed access and control into the lowest level of hardware, thus ensuring fast execution and good performance, when the application is time critical (high data rates, etc).

A drawback of the language stems from its inherent advantages: In order to ensure the fastest possible performance, the overhead and the self control of the code is minimal, resulting in errors, many of which appear at run time and are quite difficult to debug. Moreover, the freedom that it gives to the programmer, has as a counterbalance a relatively complex syntax and complicated rules and dependencies between the objects of the application, which sometimes require extensive experience from the software developer in order to build large comprehensive and easily maintainable applications.

2. Java

Java was created out of an effort of the software community to overcome the inherent difficulties of the C++. Although it is a fully object oriented language (actually more object oriented than C++; nothing can exist outside a class, unlike C++ where global variables and functions are allowed), it has several unique features such as:

- Simplified dependency rules between the different entities of the application, which result to much easier software development and maintenance.
- More overhead and checking rules, which avoid common problems of the C++ language, such as bounds checking, memory overflows and proper objects use.

- Platform interoperability. The Java applications are not written with a specific platform in mind. The output of the code compilation is a binary code which communicates with the specific hardware on which it is supposed to run, via Hardware Abstract Layers. The Java program is actually interpreted during runtime.

The main disadvantages of the language stem from its features. The extensive overhead, procedures such as the *garbage collector*, which run at unpredicted times and for unpredicted time lengths, as well as the slow execution rates due to the runtime interpretation of the binary code, raise serious doubts about the ability of the language to support the core functions of the platform, which are time-critical and require a minimum acceptable performance.

However, although rather inappropriate for the physical and data layers, Java could be a good candidate for the application layer. This layer will need to deal with constantly changing QoS settings, making the development of the applications using a less dynamic language such as C++ very difficult. Furthermore, since Java was developed with the Internet in mind, it is designed to allow easy adaptation between platforms with different capabilities.

H. MULTITHREADED PROGRAMMING

A typical software radio application will most probably involve multiple transmission and reception channels operating simultaneously. The best and most reasonable way to achieve the parallel execution of the routines serving each one of the channels, especially in a multitasking environment such as Windows where precise timing control of the execution of the code is impossible, is to use multithreaded programming.

A *thread* as the word implies, is an autonomous path of code execution. Every application is a large thread. Within this master thread, multiple other

threads can be launched, all of them running simultaneously (technically speaking they run in turns, consuming a bit of the processor time, but to the user they seem to run simultaneously) and – most important – sharing the same data segment with the launching application.

The 32 bit versions of the Windows operating system support multithreaded operation. Moreover, the last generation of the Intel Pentium 4 processors is specially designed for multithreaded applications.

The problem that multithreaded programming has to solve is how the several threads communicate and how they signal events to one another. As we shall see in Chapter 4, every time the communication gets active, a master thread is launched, which in turn launches one thread per active channel. However, how will the master thread know when the channel threads are done with their work, in order to proceed further?

The above problem has two solutions:

a) Through a global variable: The thread that wants to signal the change of state, modifies the value of a global variable. The monitoring thread constantly reads the value of the variable and when it changes, the thread takes appropriate action. Perfect you will say. Well, not exactly. This method has some drawbacks, the most important of which is that as the monitoring thread continuously monitors the value of the global variable, it consumes unnecessary processor cycles, thus slowing the execution of other threads. This bottleneck may limit the performance of the application in the case of high data rates. So, in order to overcome the problem, inevitably we pass to the second solution which is …

b) Through events. A window event is a flag which has two states: set and reset. When the monitoring procedure has to wait for a signal, it executes the order *::WaitForSingleEvent()*. By executing this command, it falls in an idle state, where it stays in a suspended mode and does not consume any processor time. When the signaling event needs to signal a change of state, it sets the

appropriate event. This action wakes the monitoring thread, which continues the execution of its code.

I. FROM HERE …

This chapter has outlined several of the technologies used in the design and implementation of a software defined radio platform. By now the user has acquired a pretty good idea, although not detailed admittedly, about the principles of operation of the equipment we are using in this thesis.

Now, it is time to see the actual equipment, have an inside look at its capabilities, the aforementioned principles that it incorporates and how we can take advantage of them in order to control the hardware for our purposes of communication. This is the objective of the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

III: DESCRIPTION OF THE HARDWARE

A. INTRODUCTION

In the previous chapter we set all the necessary theoretical background required to understand the principles of operation of a software radio transceiver. Now it is time to see how these principles are applied in practice. In the current chapter we will describe the architecture and the characteristics of the hardware we are going to use. We will follow the signal flow from the host computer memory to the transmitter output and from the receiver input to the host computer memory. We will see how the signal is processed at the several stages of the transceiver and how we can intervene and program the desired output of this processing.

At the end of this chapter, the reader will be ready to proceed to the next chapter, which is actually the description of all the work that has been done during this thesis.

B. HARDWARE CHARACTERISTICS

The hardware used for this thesis is the WaveRunner 253 Plus PCI high performance programmable transceiver, manufactured by Red River, Richardson TX. Its main characteristics are:

- Industry Standard PCI Form Factor
- 40 MHz Analog I/O Bandwidth (0 to 40 MHz)
- 8.6 MHz Maximum Signal Bandwidth
- Up to 8 Transmit and Receive Channels
- Up to 90 dB Linear Dynamic Range
- PCI Bus Master With Auto DMA Feature

- 32/64-bit and 33/66 MHz PCI Support
- Windows / Linux Drivers

A picture of the transceiver is shown in Figure 3-1. A detailed block diagram of the device is shown in Figure 3-2.

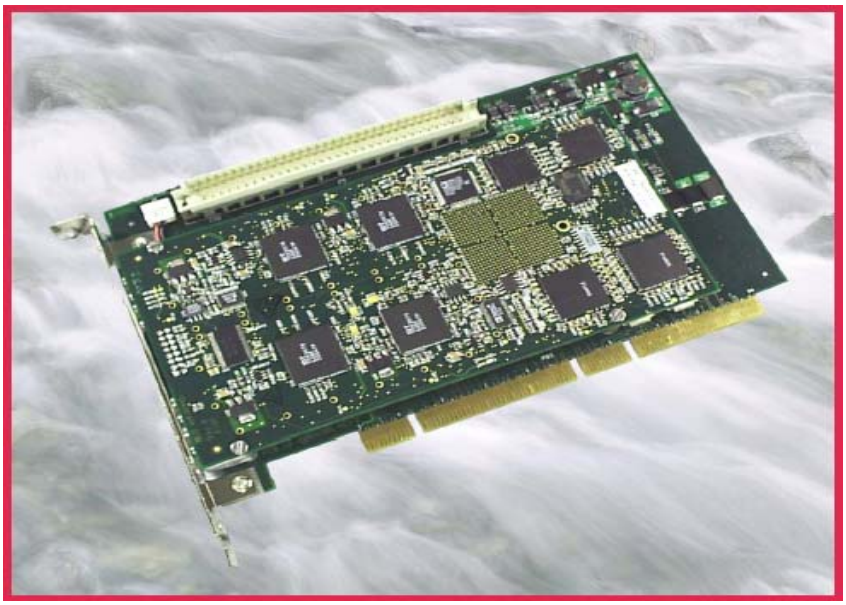


Figure 3-1. WaveRunner 253 PCI programmable SDR transceiver (From Ref. 7).

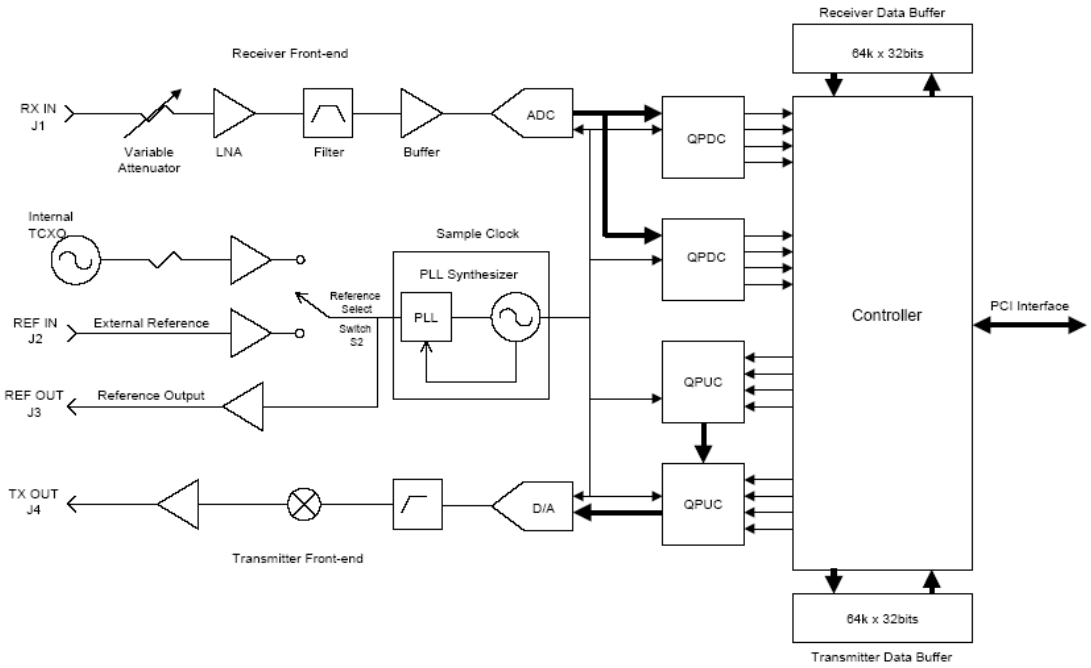


Figure 3-2. WaveRunner 253 Plus detailed block diagram (From Ref. 7).

The device consists primarily of 3 sections:

- The transmitter section
- The receiver section
- The memory controller section

The function of each one of the above sections will be described analytically in the following sections.

C. TRANSMITTER DATAPATH

The transmitter is comprised of four elements:

- The Transmitter Data Buffer
- Two Dual Quad Programmable Digital Upconverters (QPUC)
- One D/A Converter
- The Transmitter Front-End

A diagram of the transmitter datapath is shown in Figure 3-3. Detailed description of the above elements follows.

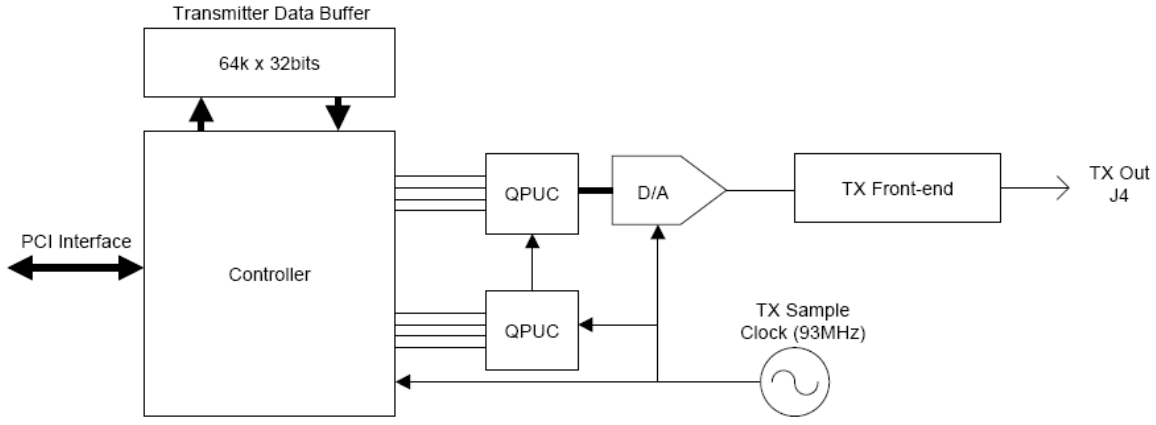


Figure 3-3. WaveRunner 253 Plus Transmitter datapath (From Ref. 7).

1. Transmitter Data Buffer

The transmitter has a dedicated 256-kbyte buffer capable of storing 64 ksamples of complex data (16-bit I, 16-bit Q). The buffer acts as a first-in, first-out (FIFO) to perform data rate translation between the PCI bus and the QPUC inputs. The buffer operates as a multi-queue FIFO that can be organized into a variety of configurations from a single large queue occupying all 256 kbytes to eight smaller queues of varying size. This flexibility provides the user ultimate control over data flow and can be used to tailor the memory to different channel rates and bandwidths.

2. Dual Quad Programmable Upconverter (QPUC)

Two Intersil ISL5217 QPUC chips are used to convert digital baseband data into modulated or frequency translated digital samples. The QPUC can be configured to create any quadrature amplitude shift-keyed (QASK) data modulated signal, including QPSK, BPSK, and M-ary QAM. The QPUC can also be configured to create both shaped and unfiltered FM signals. The QPUC performs the compute intensive tasks of tuning, filtering, resampling, interpolation and gain control.

As shown in Figure 3-4, two QPUCs in cascade provide up to eight independent data channels. The final output of the cascaded pair interfaces directly with the D/A converter to create a multi-channel analog waveform.

The key performance parameters of the QPUCs are:

Parameter	Value
Input Sample Rate Resolution	330×10^{-9} Hz
Interpolation Rate Range	16 to >65536
Max Individual Channel Bandwidth	3.5 MHz (60 dB stopband)
Max Individual Channel Data Rate	5.8 Msps (complex)
Max Poly-channel Bandwidth	8.6 MHz (60 dB stopband)
Max Poly-channel Data Rate	17.2Msps (complex)
Resampler Type	Non-integer
Filter Bandwidth Range	<1 kHz to 8.6 MHz
Maximum User FIR Taps	256
Gain (Attenuation) Range	144 dB
Carrier Tuning Accuracy (93 Msps)	0.02 Hz
Output Sample Rate	93 Msps

Table 3-1. Key performance parameters of the QPUCs

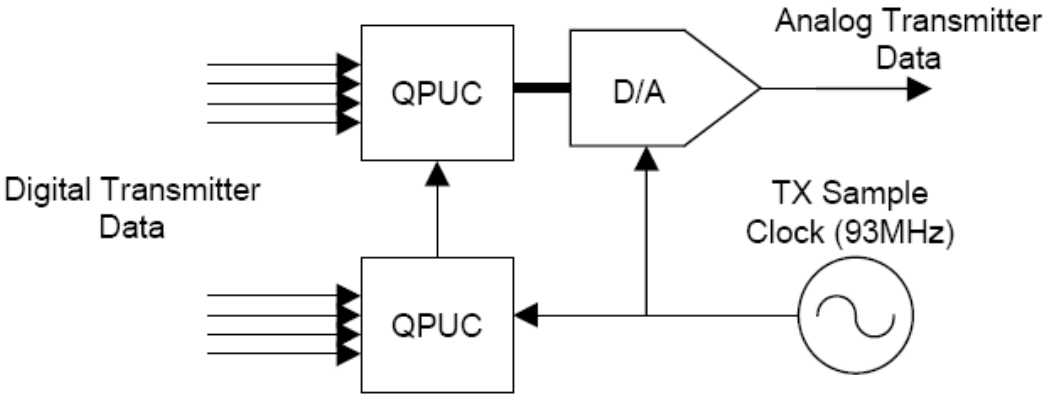


Figure 3-4. QPUC implementation (From Ref. 7).

A functional block diagram of the QUPCs is shown in Figure 3.5.

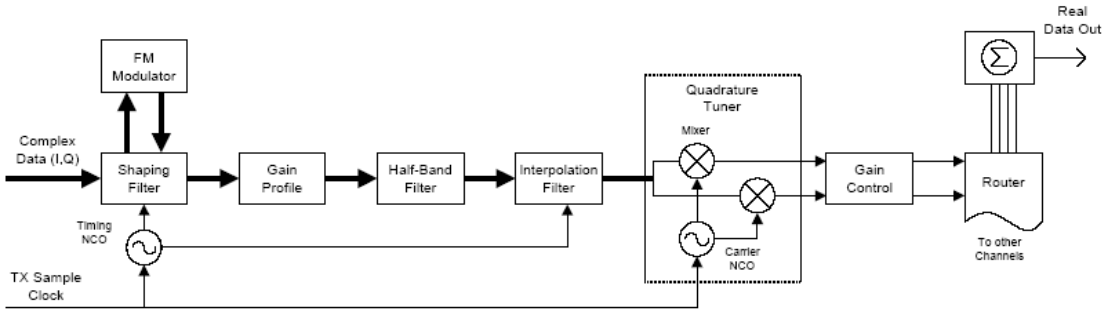


Figure 3-5. QPUC channel functional diagram (From Ref. 7).

The function of the several stages is described below:

Input Data. The QPUC operates on a 32-bit (16-bit I and 16-bit Q) complex input sample stream. Data is transferred form the transmitter data buffer at a rate determined by the timing NCO. In FM mode, the QPUC uses the I data path to process frequency samples.

Timing NCO. The QPUC features a 48-bit timing NCO driven interpolator. This high precision interpolator enables either an integer or non-integer relationship to be established between the input and output sample rates. This feature provides for the selection of almost any input sample rate, even with a fixed 93-MHz output.

Shaping Filter. The shaping filter tailors the complex symbol shape via a user configured FIR filter of up to 256 taps in length. The number of user taps available is dependent on the ratio of output to input rates, the higher the ratio the more taps available. The shaping filter must be a lowpass implementation and provide sufficient rejection to suppress images generated by subsequent interpolation stages.

FM Modulator. Each QPUC channel contains an FM modulator that can be configured to operate before the shaping filter to bandlimit the FM output, or after the shaping filter to provide shaped symbols into the FM modulator. In bandlimited FM mode the I channel is used to directly modulate the channel carrier, in shaped pulse mode the output of the shaping filter is used to frequency modulate the carrier.

Gain Profiler. The gain profile function is currently unavailable for use by the transmitter.

Halfband and Interpolation Filter. The QPUC provides a fixed 11-tap halfband filter and a timing NCO driven interpolator for data resampling and rate conversion up to the final output sample rate.

Carrier NCO/Quadrature Tuner. The carrier NCO provides sub-Hertz tuning resolution for each channel output using a 32-bit phase accumulator. It features a pre-set phase offset and synchronization capability. The carrier NCO is modulated with the digital input data using a quadrature mixer. The real component of the quadrature mixer with the form $I\cos(\theta)+Q\sin(\theta)$ is fed through the gain control section to the D/A converter.

Gain Controller. Each channel contains a gain control section that provides for attenuation of the transmitted output over a range from 0.0026 to 144 dB. A user-commanded scaling multiplier implements the gain control function using a 12-bit gain value and 3-bit shift code.

Channel Summer. The four QPUC channel outputs are combined in the Channel Summer along with the cascaded QPUC output to create a real eight-channel composite data stream for the D/A converter.

The two Intersil ISL5217 QPUC chips are labeled QPUC-A and QPUC-B for reference in the memory map. The user has access to all control and data functions through the *"QPUC-A" Command/Status Interface* and *"QPUC-B" Command/Status Interface* registers. Access to the coefficient and gain memories in the QPUC are performed using the indirect addressing method described in Ref. 8.

3. Digital-To-Analog Converter

The QPUC digital output drives a 14-bit D/A converter to generate an analog signal. The D/A converter is operated at a fixed rate of 93 MHz. The

primary D/A performance parameters are listed in the following table:

Parameter	Value (typical)
Resolution	14 bits
Sample Rate	93 Msps
SFDR	63 dBc
SFDR (4 MHz window)	80 dBc
SNR	70 dB

Table 3-2. DAC performance parameters

4. Transmitter Front-End

The D/A output passes through a 9-pole Chebyshev 40 MHz lowpass reconstruction filter prior to a final buffer amplifier. The functional diagram of the front-end is given in Figure 3-6:

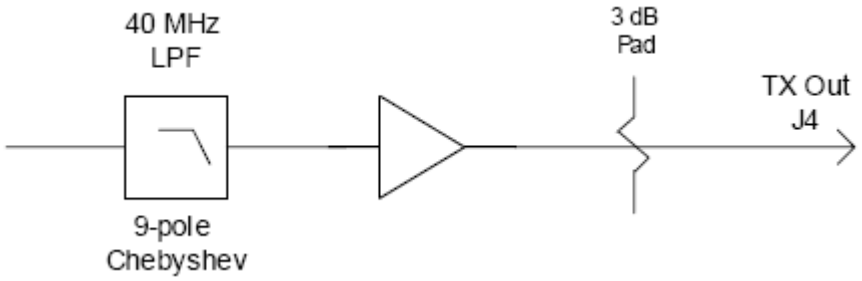


Figure 3-6. WaveRunner transmitter front-end (From Ref. 7).

The main parameters of the transmitter are described in the table below:

Parameter	Value
Sample Clock Rate	93 MHz
Phase Noise (Internal Sample Clock)	–80 dBc/Hz @ 10 kHz offset
Internal Reference Clock Stability	+/– 2.5 ppm (@10 MHz, –20 to +70 C)
Spur Free Dynamic Range	75 dB
Frequency Range (3 dB)	0.1 to 40 MHz
Full-scale Output Power (50 Ohms)	–12 dBm

Table 3-3. Transmitter main parameters

D. RECEIVER DATAPATH

The receiver is comprised of four primary elements:

- The Receiver Front-End
- One A/D Converter
- Two Dual Quad Programmable Digital Downconverter (QPDC)
- The Receiver Data Buffer

A diagram of the receiver datapath is shown in Figure 3-7. An explanation of the operation of each stage of the receiver, is given in the following paragraphs.

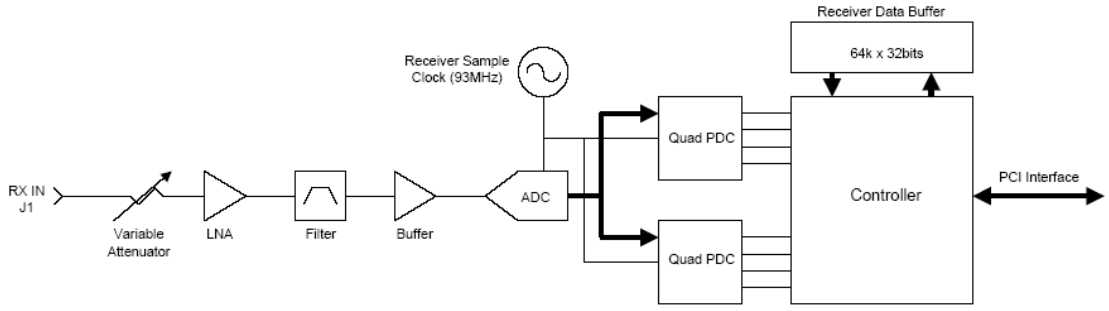


Figure 3-7. WaveRunner 253 Receiver Block Diagram (From Ref. 7).

1. Receiver Front-End

The receiver front-end amplifies and filters the analog input signal in preparation for A/D conversion. The front-end is made up of a digitally controlled attenuator, low-noise amplifier (LNA), and a filter element.

The attenuator is commanded over the PCI bus using the *Attenuator Power Control* register. The 8-bit control value can be varied from decimal 255 (no relative attenuation) to 0 (>70 dB relative attenuation). The attenuator can be used as part of software automatic gain control (AGC) circuit to control the input

level to the A/D converter. the attenuator provides precise amplitude control over the first 20 dB of operating range. The attenuator curve begins to drop steeply after the 20 dB point.

The LNA is used to set the noise figure of the receiver and help drive the input to the A/D converter. The added amplification eases the signal power requirements for systems interfacing to the receiver.

The filter is implemented as a 7-pole Chebyshev lowpass filter with a cutoff frequency of 40 MHz. The lower frequency limit of 3 MHz is dictated by the RF transformer used to couple the signal into the A/D converter.

The key parameters of the RF front-end are:

Parameter	**Value**
Analog Attenuation Range	20 dB (precision) 70 dB total
Maximum Input No damage	+15 dBm (minimum attenuation)
Sample Clock Rate	93 MHz
Phase Noise (Internal Sample Clock)	–82 dBc/Hz @ 10 kHz offset
Internal Reference Clock Stability	+/– 1.5 ppm (@10 MHz, –20 to +70 C)
Linear Dynamic Range (1 MHz bandwidth)	90 dB
IMD Rejection	75 dB (typical)
Frequency Range (3 dB)	0.1 to 40 MHz
Full-scale Input Power (50 Ohms)	–14dBm (minimum attenuation)
Noise Figure	7.9 dB
Input Third Order Intercept (IIP3)	+6.2 dBm (minimum attenuation)
IMD Rejection	80 dB (typical)

Table 3-4. RF-Front-end key parameters

2. Analog-To-Digital Converter

The receiver signal is sampled using a 14-bit A/D converter that can be used in either bandpass or baseband operating mode. The A/D operates at a fixed rate of 93 MHz with a resolution of 14 bits.

3. Dual Quad Programmable Digital Downconverter (QPDC)

Two Intersil ISL5216 QPDC chips perform the digital receiver tuning and filtering functions. The digital downconverter is one of the key receiver signal conditioning components in a software radio system. It can be configured directly from user application code to form carrier and symbol recovery loops as well as other critical demodulation functions as part of software defined radio.

The dual QPDCs provide up to eight independent data channels as shown in Figure 3-8. The QPDC accepts raw sample data directly from the A/D converter and performs the compute intensive tasks of tuning, filtering, decimation, gain control, and re-sampling.

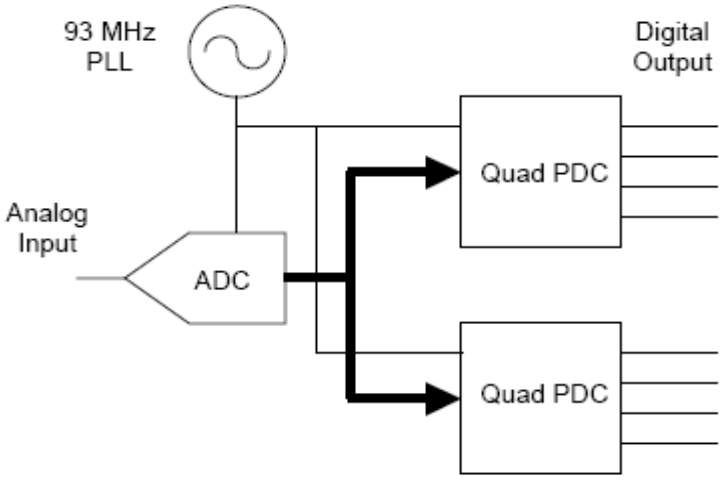


Figure 3-8. WaveRunner 253 QPDC configuration (From Ref. 7).

Some of the key performance parameters of the QPDC are listed in the following table:

Parameter	Value
Input Sample Rate	93 Msps
Carrier Tuning Accuracy (93Msps)	0.02 Hz
Decimation Rate Range	4 to more than 65536
Max Individual Channel Bandwidth	3.5 MHz (60dB stopband)
Max Individual Channel Data Rate	5.8 Msps (complex)
Max Poly-channel Bandwidth	8.6 MHz (60dB stopband)
Max Poly-channel Data Rate	23.2 Msps (complex)
Filter Bandwidth Range	<1 kHz to 8.6 MHz
Maximum User FIR Taps	256 typical, 384 max
Maximum FIR Filter Rejection	110 dB
AGC Range	96 dB
Output Sample Rate Resolution	1.29×10^{-9} Hz
Resampler Type	Integer and Non-integer
Output Formats	I, Q, Mag, Phase, Frequency

Table 3-5. Key performance parameters of the QPDCs

A functional block diagram of a single QPDC channel is shown in Figure 3-9. Each channel implements a sub-band tuning architecture by tuning a narrow filter to a selected point in the A/D converter passband.

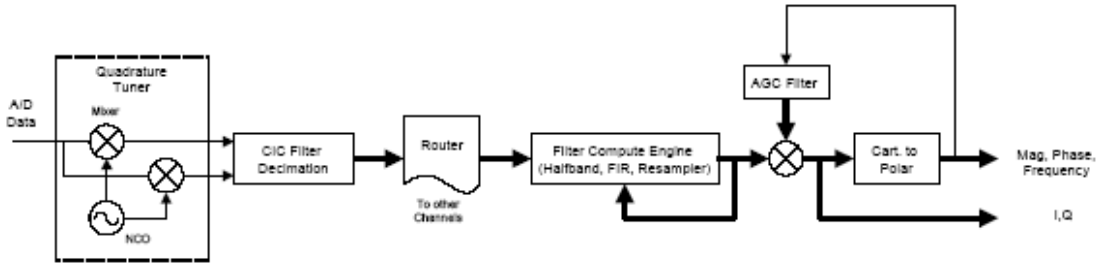


Figure 3-9. QPDC Block diagram (From Ref. 7).

The major elements of the QPDC channel are:

- Input Level Detector (not shown)
- Carrier Numerically Controlled Oscillator (NCO)/Quadrature Tuner
- Cascaded Integrator Comb (CIC) Filter

- Filter Compute Engine/Resampler
- Timing (symbol) NCO
- AGC
- Cartesian-to-Polar Converter (Magnitude, Phase, Frequency)

The following paragraphs provide a brief overview of QPDC functionality:

Input Level Detector. The input level detector monitors data as it enters the QPDC and provides three averaging modes for determining the input signal magnitude in the A/D converter. The three modes supported are integration, leaky integration, and peak detection. The information from the level detector can be used to adjust the analog gain as part of a software AGC loop.

Carrier NCO/Quadrature Tuner. The tuning function is implemented using an NCO and quadrature mixer. The NCO phase accumulator is 32 bits wide, allowing for sub-Hertz tuning accuracy and very low (–115 dB) spurious response.

CIC Filter. The CIC filter is the first signal conditioning element in the QPDC processing chain. The overall shape of the processing band is determined through a combination of the CIC section and components of the filter compute engine: half band, FIR, and resampler.

The CIC filter has a $\sin(x)/x$ characteristic as shown in Figure 3-10. It is an extremely efficient filter and is used to extract a band of interest and drop the sample rate prior to precision filtering in the Filter Compute Engine section.

Filter Compute Engine (FCE). The purpose of the FCE section is to fine tune the processed signal spectrum created by the CIC filter. The FCE provides a variety of user-configurable filters including a series of fixed half-band filters

and a user-programmable finite impulse response (FIR) filter. The frequency response of the built-in halfband filters is shown in Figure 3-11. The FCE also provides for non-integer resampling using a polyphase filter in concert with a precision timing NCO. The FCE has a built-in controller used to chain together filter and decimation stages to create a precisely conditioned signal with an exact shape, bandwidth, and sample rate. An example of a composite filter created from the CIC and FCE combination is shown in Figure 3-12.

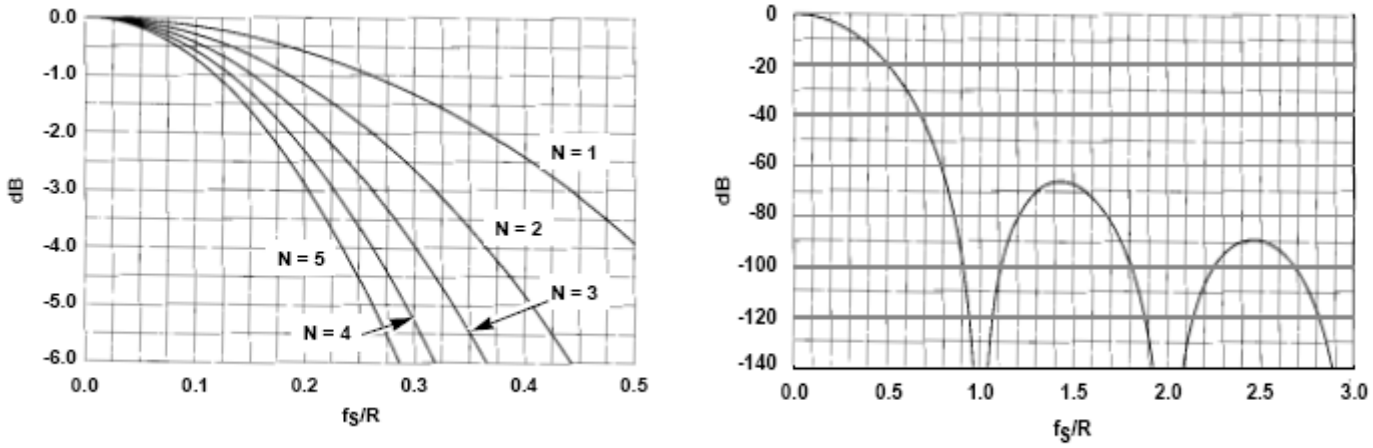


Figure 3-10. CIC characteristics. (a) Passband rolloff ($N =$ Number of stages, $R =$ decimation factor, $f_s =$ sampling frequency). (b) 5^{th} order ($N = 5$) CIC filter response (From Ref. 8).

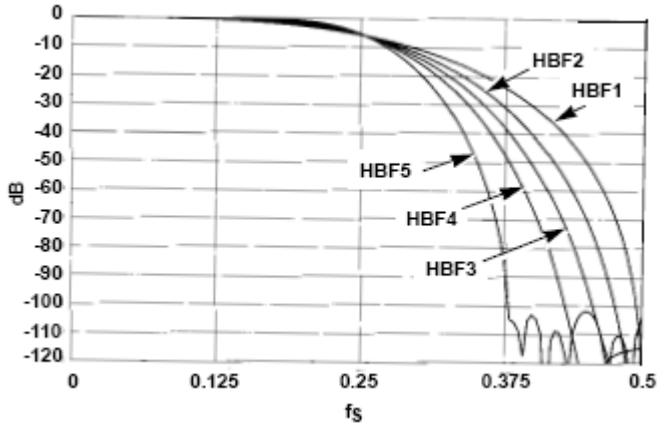


Figure 3-11. Frequency response of the built-in halfband filters (From Ref. 8).

Timing (symbol) NCO. The QPDC features a high resolution (56-bit) timing NCO that, when used with the resampler, provides the user with the capability to generate precise output sample rates independent of the QPDC input sample rate.

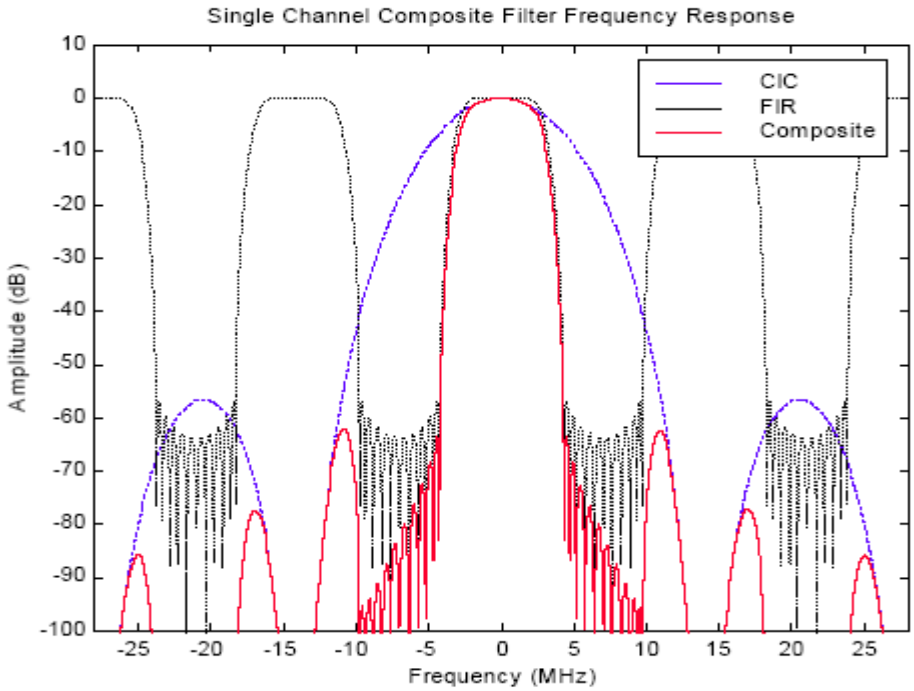


Figure 3-12. Composite filter example (CIC + FIR) (From Ref. 7).

AGC. A full featured AGC loop is built into the QPDC for high performance signal level control. The AGC loop can be used to extract small signals from noise after large signals and out-of-band noise have been filtered by the CIC and FIR filter sections. The AGC loop operates on 24-bit data emanating from the FCE to provide up to 96 dB of dynamic range.

Cartesian-to-Polar Converter (Magnitude, Phase, Frequency). The last section of the QPDC contains a Cartesian-to-Polar (CTP) converter. The CTP converts data from complex I and Q format to a polar form made up of magnitude and phase. Both data formats are available to the user to save software overhead in converting between the two systems. The QPDC also

contains a frequency discriminator that can be used to directly output frequency measurements.

QPDC Command/Status. The two Intersil ISL5216 QPDC chips are labeled QPDC-A and QPDC-B for reference in the memory map. All the QPDC configuration registers are user accessible through the *QPDC-A Command/Status Interface* and *QPDC-B Command/Status Interface* registers (Ref. 8).

4. Receiver Data Buffer

The receiver has a dedicated 256 kbytes buffer capable of storing 64 ksamples of complex data (16-bit I, 16-bit Q). The buffer acts as a FIFO to perform data rate translation between the QPDC outputs and the PCI bus. The buffer operates as a multi-queue FIFO that can be organized into a variety of configurations from a single large queue occupying all 256 kbytes to eight smaller queues of varying size. This flexibility provides the user ultimate control over data flow and can be used to tailor the memory to different channel rates and bandwidths.

E. CONTROLLER

The controller performs all local command and control functions while also acting as an interface to the host computer. It communicates with the host over an interface compliant with the PCI Local Bus Specification (66 MHz). The interface supports both 32- and 64-bit transactions operating at either 33 or 66 MHz.

The controller occupies 2 Mbytes of memory space accessed from a single base address register and is connected to PCI Interrupt A on the bus. The PCI bus serves as the primary control and data interface to the card. The local

controller can operate as a master or target, with the ability to automatically initiate DMA transfers of data between the transceiver buffers and host memory.

Interrupt Control / Status. A comprehensive set of interrupt enable and status registers are available to support data movement and alert the host to error conditions. Interrupt conditions are reported through the *Interrupt Status* and *Rx/Tx FIFO Interrupt Status* registers. All interrupt flags are "trapped" and held until an interrupt status read occurs which clears the flags in the respective register.

Buffer Configuration. Separate 256-kbytes (64-bit by 32k) buffers are dedicated to the receive and transmit datapaths. The buffers can be accessed directly through 32- or 64-bit single or burst transfers over the PCI bus.

Memory Area Definition. Both the receive and transmit buffers can be partitioned as one to eight distinct memory areas that are assigned to a corresponding number of QPDC and QPUC channels. Each Memory Area looks like an independent FIFO to the PCI bus interface. The Memory Areas can be arbitrarily sized on 8-byte boundaries to load balance channels with different data rates. The controller maintains read and write pointers to keep track of the next address to be accessed.

Channel Assignment. The QPDCs and QPUCs can be configured to process eight channels separately or to polyphase multiple (≥2) channels together. If they are configured as eight independent channels, the controller transfers data between each channel and the uniquely assigned Memory Area. When they are configured to polyphase multiple channels together, data from groups of channels need to access the same Memory Area.

DMA Transfers. The controller can transfer data to/from the host memory using manual or automatic direct memory access (DMA). Manual DMA transfers

require the user to initiate the DMA transfer by writing to several registers. Manual DMAs are typically initiated when a Memory Area FIFO generates an over threshold interrupt. When Automatic DMA is used, the controller monitors the status of the memory FIFO thresholds. When a threshold is exceeded, the controller automatically initiates a DMA transfer to/from a user specified memory location. Automatic DMA also has a provision for notifying the host process via an interrupt that the previously specified number of DMA transfers have been completed.

DMA Chaining. The Automatic DMA function is supplemented by a DMA chaining feature that allows the controller to transfer data to/from a circular buffer in host memory without processor intervention. The *Auto DMA Block Count* and *Auto DMA Address* registers remain static when this feature is enabled. An interrupt is still issued when transfer of the final data block completes, but the controller will continue to transfer data to/from the next sequential address in host memory instead of loading a new address.

More extensive description of the the channel configuration will follow at the next chapter, because the details of these characteristics are the main area of interest for the implementation of our software.

F. FROM HERE …

The current chapter has outlined a brief description of the characteristics of the hardware we are using. Indeed the capabilities of the hardware are vast. Here we have described only a part of them, only those that are going to be useful to our work.

Now, we are ready to proceed to the next chapter which constitutes the actual work that has been performed: the description of the software developed by the author.

IV: DESCRIPTION OF THE SOFTWARE

A. INTRODUCTION

In the previous chapters we examined some aspects of the theoretical background of the Software Defined Radio technology, as well as the specific architecture of the hardware we used in our effort to implement the SDR datalink.

Now it is time to describe the actual work that was done during this thesis. This work actually consisted of two parts:

- Configuration and manipulation of the hardware
- Data organization in order to achieve successful and meaningful communication.

The first one of the above two parts was actually the most rigorous one. It can be analyzed in the following tasks:

- Channel configuration
- Hardware initialization – configuration
- Data exchange administration

In order to implement the above described functionality, significant amount of code was written in C++, using the *Microsoft Visual C++* language, which constitutes a part of the *Visual Studio .NET* programming suite. The channel configuration was performed using a dedicated configuration tool provided by the manufacturer.

The following paragraphs describe in detail all the work that has been carried out.

B. HARDWARE LIBRARY AND DATA TRANSFER MODE

The hardware we used was accompanied by a driver for the Windows operating system and a library of functions, through which the application could access and program the card. These functions covered a variety of tasks: opening and closing the card, allocating memory, configuring the PCI address space, reading from and writing to specific card registers. The library was incorporated into the application code.

After the card has been configured, data transfer between the host computer allocated memory and the card buffers takes place automatically without user intervention, taking advantage of the *DMA chaining* mode. During this mode the user defines the parameters of the transfers (groups per channel, blocks per group, symbols per block, memory areas-limits-starting and ending offsets) and the DMA address is incremented automatically every time a symbol is transferred, until it reaches the end of the memory segment allocated to a specific channel. At that point, the DMA address is reset to its original value. In this way, the channel memory is used as a circular buffer containing two or more groups. When the card is accessing one of the groups, the host computer is accessing another one. It is obvious that the data transfer rate between the host computer memory and the storage medium (for example the hard disk) must be at least as fast as the transfer rate between the card and the host memory. This is the critical factor for the performance of the system. When a group of data has to be transferred to or from the host computer memory, the card notifies the host computer by raising the appropriate interrupt. The application responds to the interrupt and takes the appropriate action via an *Interrupt Service Routine.* A dummy routine is provided in the card library, which does practically nothing. One of the tasks of the user application is to override this function with a more meaningful one.

C. CHANNELS CONFIGURATION

The configuration of the WaveRunner channels actually consists of two parts:

- Configuration of the Upconverter (DUC) and Downconverter (DDC) integrated circuits with all the parameters of the channels: center frequencies, datarates, upconversion and downconversion rates, programming of the shaping filter of the transmitter and the FIR filters of the receiver, data exchange between the converters and the card memory, possible combinations of the DDC circuits in the case of polyphase channel organization e.t.c.
- Configuration of the main memory of the card in respect to the number of reception and transmission channels selected by the user.

Both the above tasks can be performed using the *WaveFormer II configuration utility*. This utility consists of a series of Excel spreadsheets organized in a hierarchical manner, giving to the user access to all the registers of the hardware. After the user has chosen all the desirable values, three C header files are produced, which can be uploaded to the hardware by the user application.

However, the drawback of the above procedure is that each time one parameter needs to be modified, the user must open and re-run the utility, compile and reload the files. Since the author's belief was that the true value of the software radio resides on its adaptability, the user should have a maximum degree of flexibility and should be able to change some parameters "on the fly". The flexibility that was decided to be available through the application that would be developed, was the arbitrary choice of the number of active reception and transmission channels, center frequencies, datarates and modulation types. All of the above tasks would be administered solely by software.

In order to implement the above, only the DUC and DDC configuration was decided to be performed with the use of the configuration tool, because it was fairly complicated and the time restrictions of this thesis prohibited the implementation of the above functionality in C++. On the other hand, the organization of the card memory, which would depend on the number of active channels selected by the user, as well as the administration of the data exchange between the card and the host computer, would be implemented by the C++ application.

Figures 4-1 to 4-4 show several snapshots of the configuration utility. It can be clearly seen that a multitude of parameters can be set with this utility. After all parameters are set, the user returns to the main screen and creates the header files.

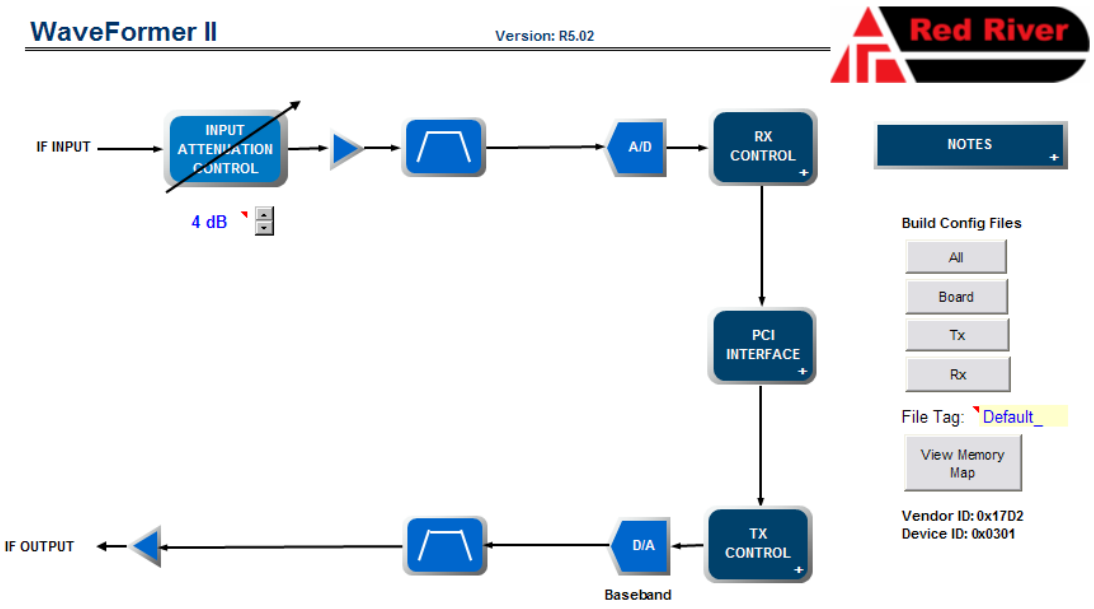


Figure 4-1. Main screen of the WaveFormer configuration Tool.

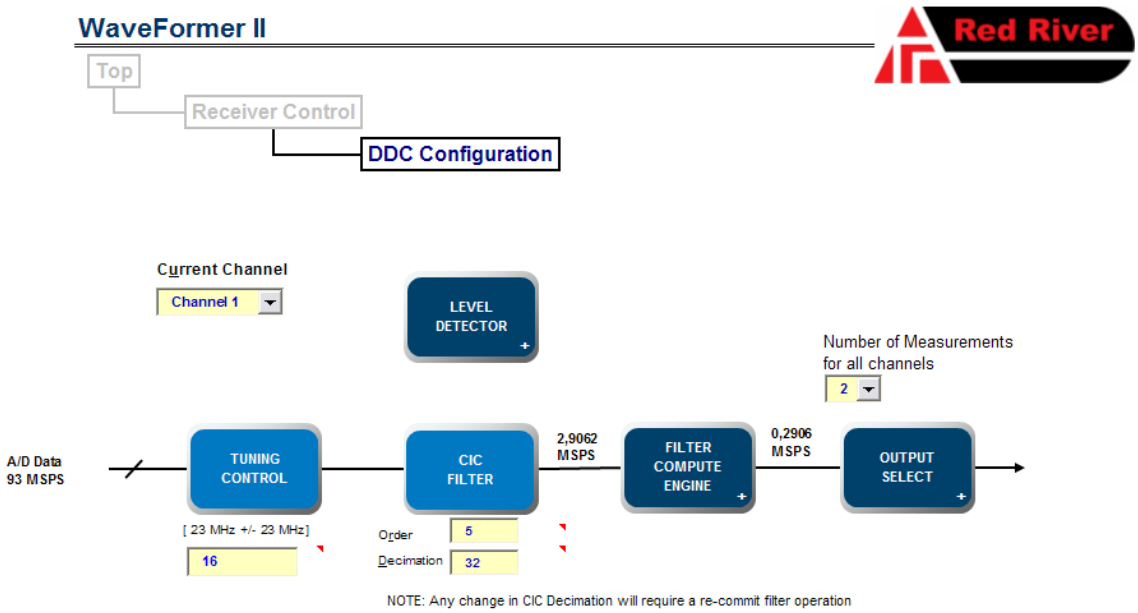


Figure 4-2. Reception channel configuration screen.

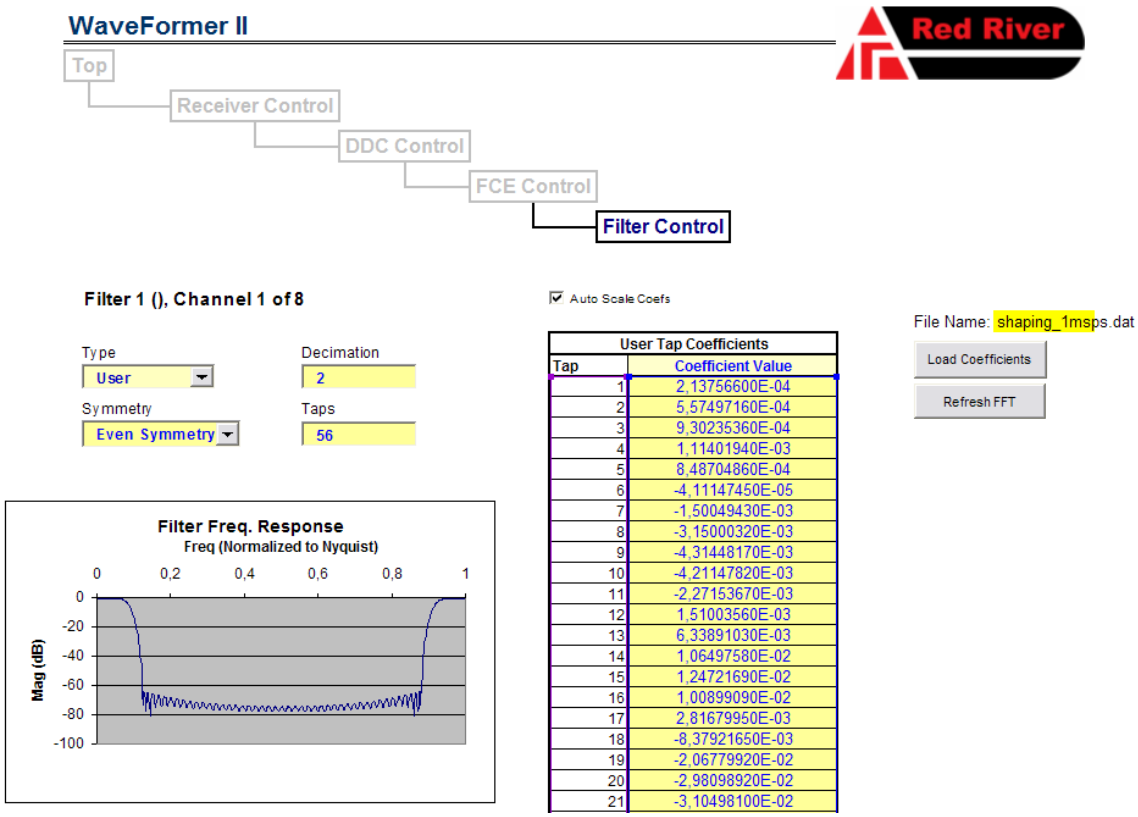


Figure 4-3. Configuration of one of the filters of the DDC FIR engine.

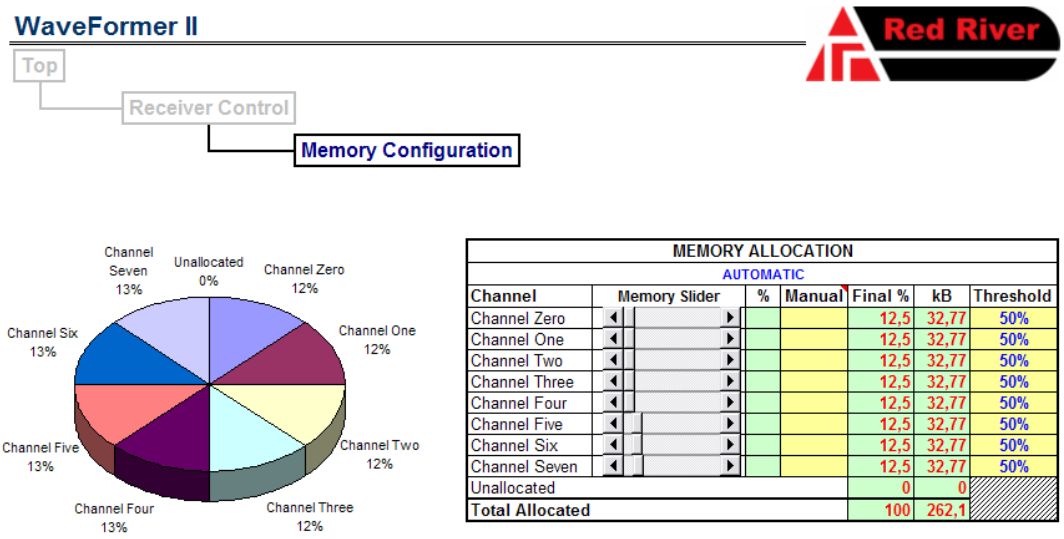


Figure 4-4. Memory organization for the transmission channels.

D. APPLICATION GUI

As we have already mentioned, an application was created by the author using the Microsoft Visual C++ language. On the top of this application lies a Graphical User Interface (GUI) which, through a series of screens, lets the user input all the desired parameters of the communications: number of active channels, center frequencies, modulation types and datarates.

Figures 4-5 to 4-7 show the basic screens of the application GUI. It actually consists of three pages:

- **Main page**: In this page the user chooses the number of active channels and the location of the configuration file (which has already been produced by the *WaveFormer* configuration tool). After configuring all the features of the active Rx and Tx channels in the following two pages, the user presses the **Start Rx/Tx** button and the card is activated starting the communication. The user can be

informed on the status and progress of each active channel, by the progress bars and text boxes at the lower half of the page.

- **Rx channels configuration page**. From this page, the user can define the following parameters for each active Rx channel: center frequency, modulation type, number of samples per symbol (more about this later) and location of the file to store the demodulated data.
- **Tx channels configuration page**. In the same manner as in the previous page, the user can define the following parameters for each active Tx channel: center frequency, modulation type, datarate, location of the file to retrieve the data to be transmitted and attenuation of the transmitted signal (in terms of its maximum possible power).

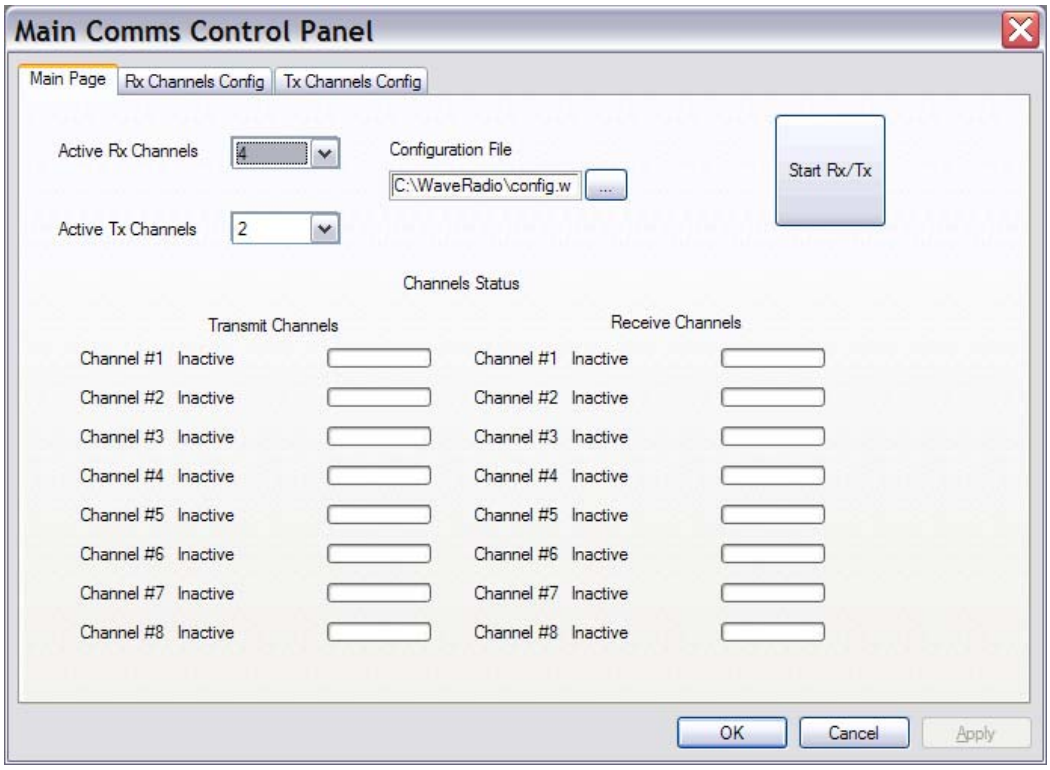


Figure 4-5. Communications Control Panel main page.

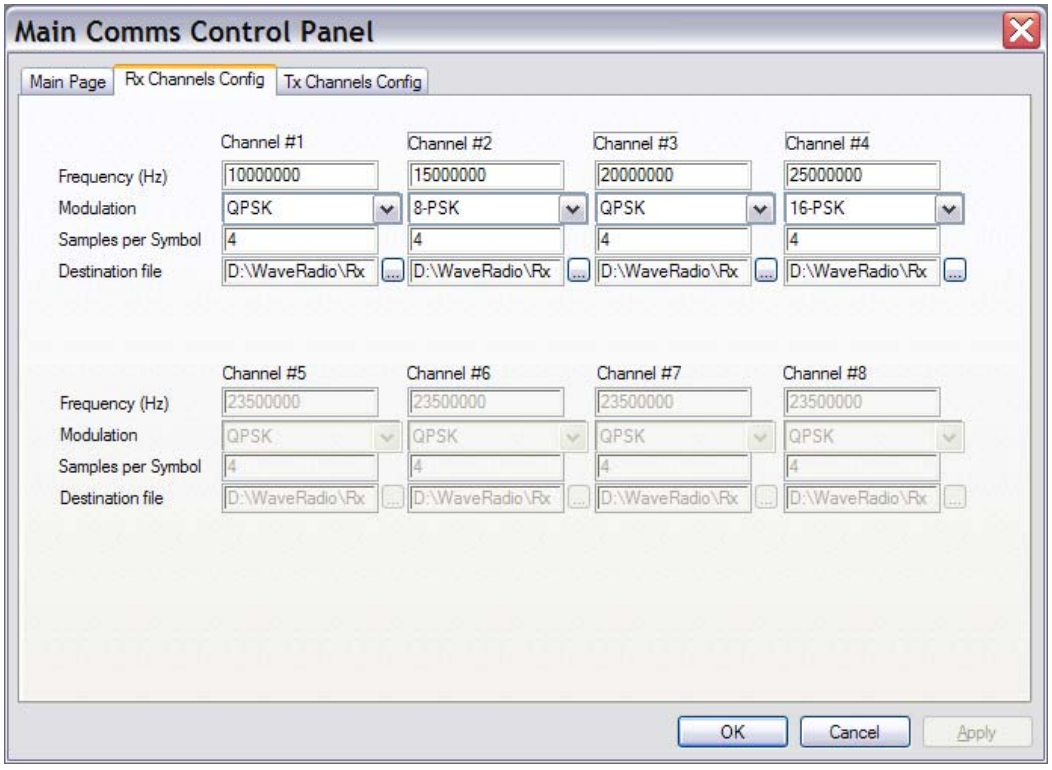


Figure 4-6. Communications Control Panel Rx channels configuration space.

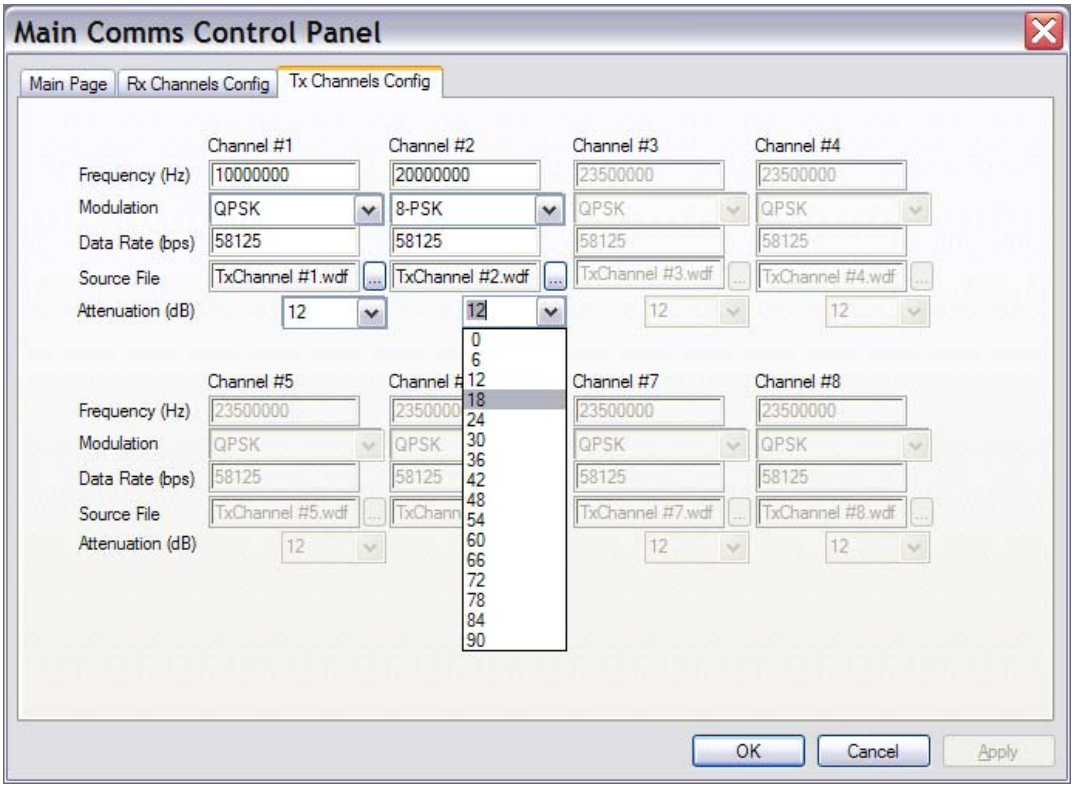


Figure 4-7. Communications Control Panel Tx channels configuration space.

E. APPLICATION ARCHITECTURE

In the previous paragraphs we described the user interface of our application: what the user sees and how he interacts with the application. However, the quintessence of the code lies on what takes place after the user has pressed the **Start Rx/Tx** button: the procedures that were developed in order to achieve successful communication and data exchange between the transceiver and the host computer. These procedures are described extensively in the following paragraphs.

1. Objects

Since the application is purely object oriented, a series of classes were created, each of them containing all the necessary functionality in order to achieve specific tasks. The main classes of the application are described in the following paragraphs. For each object, a table summarizes its main variables and functions.

WaveRunner. This class encapsulates the functionality of the card. Only one instance of this class can be created in the application using the *Singleton pattern* (Ref. 6). The class also incorporates the objects encapsulating the functionality of the channels (they will be described later in this paragraph).

Class WaveRunner	
Main Variables	
Const	
`maxChannels`	Number of maximum available channels
`rxClockFrequency`	Frequency of the receiver circuit
`txClockFrequency`	Frequency of the transmitter circuit
`blockSize`	Size in 4-byte words of a block of data

`rxBlocksPerGroup;`
`rxGroupsPerChannel;`
`rxThresholdGroups;`
`rxChannelSize;`
	Reception organization parameters

`txBlocksPerGroup;` `txGroupsPerChannel;`	Transmission organization parameters

txThresholdGroups; txChannelSize;	} Transmission organization parameters
memorySize;	Size of host computer memory allocated to the device
lDMAvAddress;	Virtual starting address of the allocated memory
lDMApAddress;	Physical starting address of the allocated memory
rxChannel[maxChannels]	Array of RxChannel pointers
txChannel[maxChannels]	Array of TxChannel pointers
rxThredsRunning	Number of active reception threads
txThreadsRunning	Number of active transmission threads
txChannelsCount;	Number of selected active transmission channels
rxChannelsCount;	Number of selected active reception channels
rxChannelsConfigured;	Number of configured active reception channels
txChannelsConfigured;	Number of configured active transmission channels
rxTxEnable	Boolean variable reflecting the status of the card
Main functions	
getInstance()	Creation of one and only new WaveRunner object using the *Singleton pattern*
Open()	Initial opening of the device
Close()	Closing of the device
Configure()	Configuration of the device
EnableRx()	Reception enable (requires configuration first)
DisableRx()	Reception disable
Enable(Tx)	Transmission enable (requires configuration first)
DisableTx()	Transmission disable
EnableRxTx()	Reception and transmission enable
DisableRxTx()	Reception and transmission disable

Table 4-1. WaveRunner class description.

The details of the functionality of the WaveRunner class functions will be described later in this chapter.

WaveRunnerChannel. This is an abstract class. It contains the main common parameters of the transmission and reception channels. It is the parent class which will be inherited to the Tx and Rx channel classes.

Class WaveRunnerChannel	
Main Variables	
`channelNumber;`	Number of channel (0-7)
`channelOffset;`	Channel offset (used for programming the DUC and DDC registers)
`frequency;`	Channel center frequency
`k;`	Number of bits per symbol (for M-PSK modulations)
`dataFileName;`	Name of the file to store or retrieve data
`dataRate;`	Channel datarate (for the Tx channels) or samples per symbol (for the Rx channels)
`offsetAddress;`	Offset address of the DDC or DUC registers.
`dataBuffer;`	Pointer to the allocated memory of the channel
`groupsTransfered;`	Number of transferred groups of data to/from the transceiver
`groupCount;`	Counter of the starting position on the channel memory to transfer data.
`terminateProcess;`	Orders the channel thread to be terminated
`threadRunning;`	Indicates if the channel thread is running
`threadReady;`	Indicates if the channel thread is ready
Main functions	
`WaveRunnerChannel()`	Dummy class constructor. No useful action is performed.

Table 4-2. WaveRunnerChannel class description.

RxChannel. It stores the functionality of a reception channel. It inherits all the parameters of the *WaveRunnerChannel* class.

Class RxChannel	
Main variables	
`groupsSaved`	Number of groups saved to the destination file
Main functions	
`RxChannel()`	Class constructor. Initializes all the class parameters. Calls `setFrequency()`.
`setFrequency()`	Sets the channel center frequency by writing to the appropriate DDC register.

Table 4-3. RxChannel class description.

TxChannel. It incorporates the functionality of a Tx channel. It also inherits all the parameters of the *WaveRunnerChannel* class.

Class TxChannel	
Main variables	
`groupsLoaded` `attenuation`	Number of groups loaded from the Tx file. Attenuation in dB of the transmitted data
Main functions	

`TxChannel()`
`setFrequency()`
 	
`setDataRate()`
	Class constructor. Initializes all the class parameters.
	Sets the center frequency of the channel by writing to the appropriate DUC register
	Sets the datarate of the channel by writing to the appropriate DUC register.

Table 4-4. TxChannel class description.

Apart from the above classes, the application uses four other classes in order to implement the GUI. The first one of them is the **CommsCtrlDlg** class. It is a *CPropertySheet* class and serves as the nesting class for the three classes, **CommsTab1**, **CommsTab2** and **CommsTab3.** These three classes are of the *CPropertyPage* type, each one of them incorporating the functionality of the corresponding page of the communication control panel. Through a series of text boxes, combo boxes, property sheets, buttons and message boxes, the above classes ensure that the user has entered the correct parameters. Also, while the card is active, the user is informed on the process and status of all the active channels.

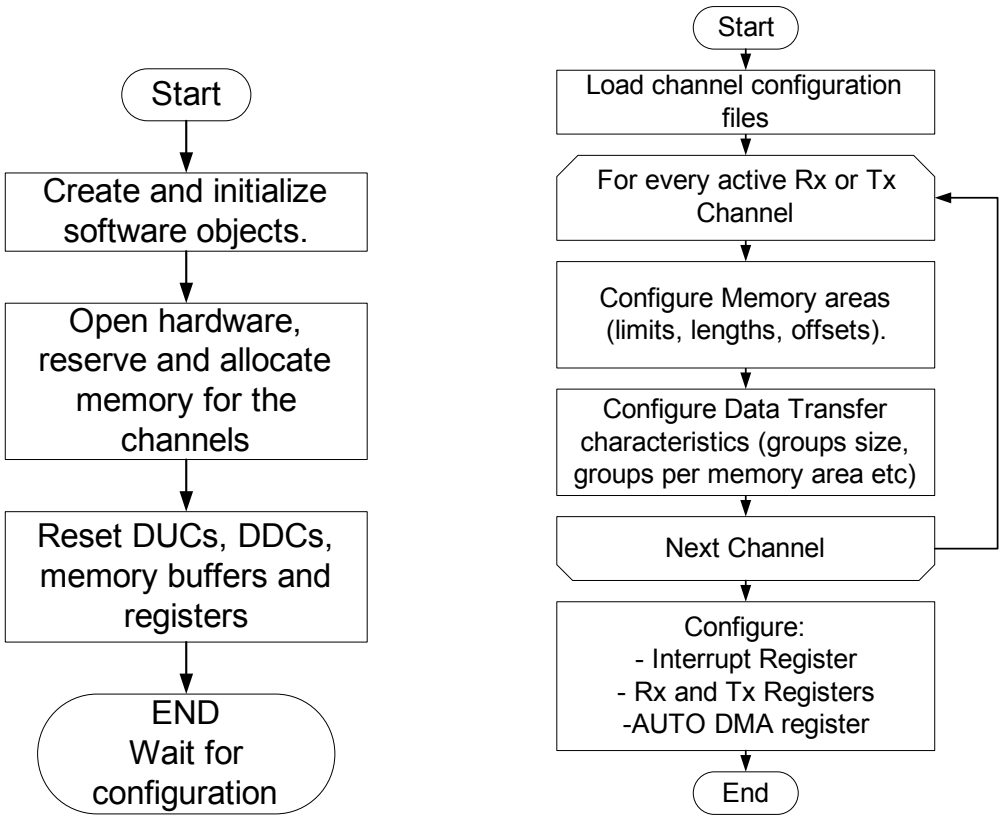
The implementation of these classes constitutes a significant portion of the application code. However, since their functionality is common C++ functionality and not directly connected to the subject of this thesis, it was considered better not to describe them analytically in this text. Their code is included in the listing of the application code at the appendix of this thesis.

2. Procedures

Hardware initialization (Figure 4-8a). When the application starts, an object of the *WaveRunner* class is created. As a part of its initialization code, the object checks to see if the card is present, by executing the *OpenWaveRunner()*

command from the library. If this is the situation, the hardware initialization takes place, by executing the *WaveRunner::Open()* function. The routine initializes the channel objects, allocates host computer memory and distributes it to the channel objects, resets the card interrupt register and DMA Transfer register, resets the DUCs and the DDCs and exits.

After that the application waits for the user to enter the appropriate parameters and hit the **Start Rx/Tx** button.



(a) (b)

Figure 4-8. (a) Hardware initialization procedure. (b) Hardware configuration procedure.

Hardware configuration (Figure 4-8b)**.** The click of the **Start Rx/Tx** button changes its caption to **Stop Rx/Tx** and invokes the master thread *mainRxTxThread().* The first thing that this thread does, is configure the

hardware by invoking the *WaveRunner::Configure()* function. The sequence of configuration tasks is the following:

- The C header files that have been created using the *WaveFormer* configuration tool, are uploaded to the card using the *ConfigureWaveRunner()* command of the library.
- The PCI configuration space is configured with the number of 4-byte words per data block, using the *WriteWRConfigSpace()* command from the library.
- For every active transmission and reception channel the following parameters are defined by writing specific values to the appropriate registers:
    - Starting address of the channel memory.
    - Number of data blocks per group of data transfer.
    - Number of groups per channel memory area.
    - Channel memory area size, starting offset, ending offset and threshold to raise an interrupt.
    - The interrupt mask register is updated in order to permit interrupts from the specific channel.
    - The receive or transmit control register is updated in order to service the specific channel
- The Auto DMA Transfer Register is updated in order to perform data transfers in the active channel memory spaces.
- The card buffers are flushed in order to delete any random data.

Interrupt Service Routine (Figure 4-9a). The application provides useful behavior to the interrupt service routine by overriding the *PMCRadioIsr0()* routine of the WaveRunner library. When an interrupt occurs, this function is called with the contents of the Interrupt Status Register passed as a parameter. The routine

checks to see if the interrupt is due to the transceiver. If this is the situation, it checks to see if the interrupt is due to a buffer abnormal condition (overflow or underflow). In this case, it updates the corresponding channel status. Otherwise, the interrupt is due to a completed data transfer. So, the appropriate channel event is set (more about this later). As a last step, the routine re-enables the interrupts and exits.

At this point we need to say that the interrupt service routine must be as fast as possible, since it is invoked hundreds or even thousands of times per second. As long as the routine is invoked, additional interrupts are disabled. That means that data corruption may occur. That is the reason why the routine only signals the channel servicing routines that they have to take appropriate action. It is the responsibility of the channel threads to take whatever action they deem necessary.

Main communication thread (Figure 4-9b). As we have already said, this thread is launched when the **Start Rx/Tx** button is pressed. The tasks that this thread performs are the following:

- It calls the *WaveRunner::Configure()* function, which performs the hardware configuration.
- For each active channel, it transfers the parameters chosen by the user, into the *WaveRunner* and the channel class variables.
- For every active channel, it launches a corresponding *rxThread()* or *txThread()*.
- It waits until all the active channels are ready to transmit or receive.
- It enables the transmission and reception circuitry of the card. More specifically:
    - It enables interrupts by writing a one to the Global Interrupt Register.

- It sets the appropriate bits of the Transmit Control and Receive Control register.
- It enables the DMA data transfer, by setting the appropriate bit of the Auto DMA Register.

- It waits until all the channels are done transmitting or receiving.
- It disables transmission and reception.
- It notifies the application that the card is no longer transmitting or receiving.

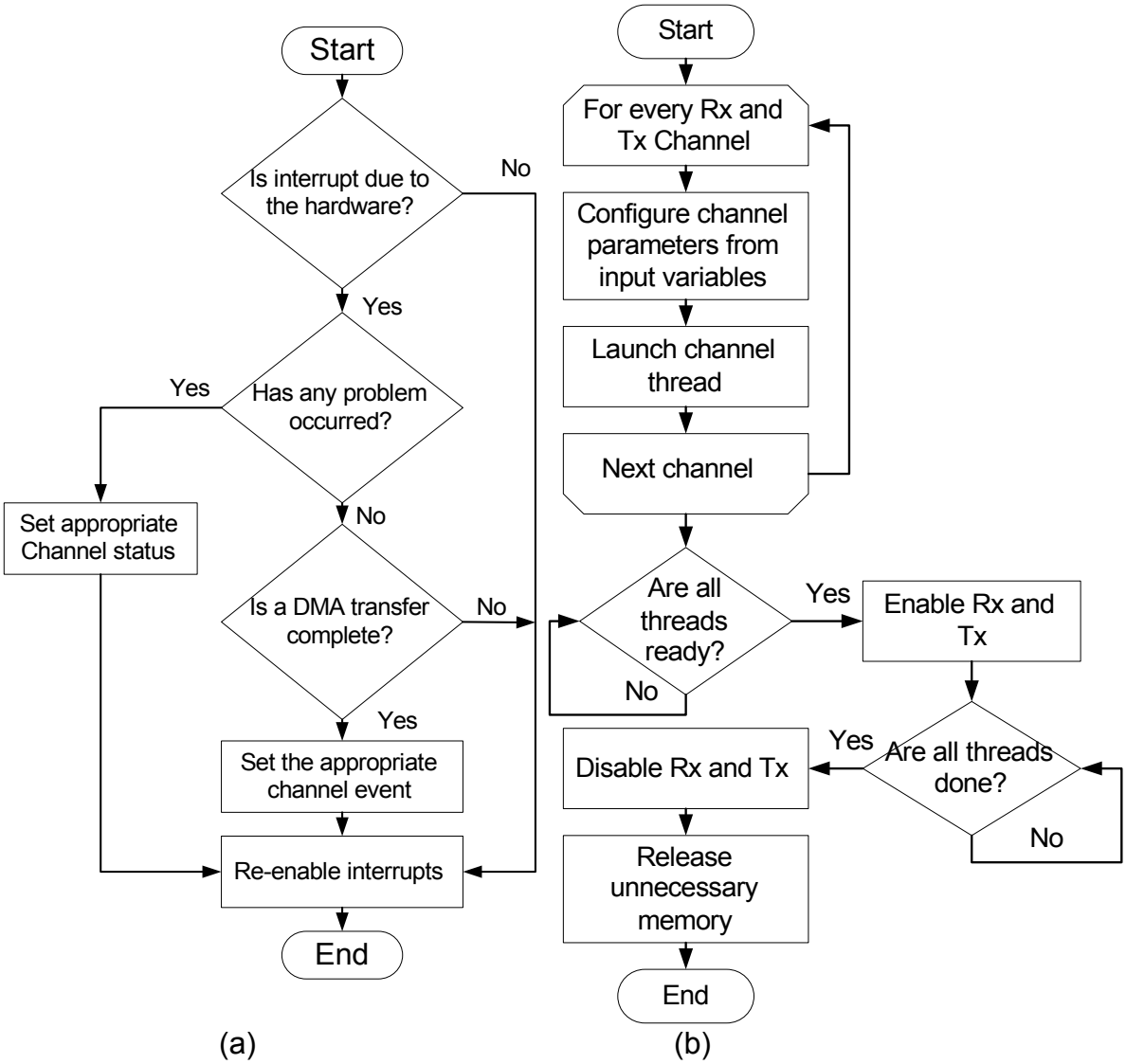


Figure 4-9. (a) Interrupt Service Routine. (b) Main communications thread.

We need to say that, at the two occasions when the master thread waits for the channel threads to signal an event, the signaling is accomplished via the setting of two events, the *allChannelsReady* event, in order to signal that all channels are ready for transmission and reception and the *allChannelsDone* event, which signals that all channels are done.

Just before the routine exits, if the user has not disabled reception or transmission in the meantime, the thread posts a *WM_PROCESS_FINISHED* user defined message to the application GUI, in order for the **Start Rx/Tx** button caption to return to its original state.

Tx thread (Figure 4-10a). This thread is launched by the master Rx/Tx thread, once for every active transmission channel.

The first task of the thread is to create the baseband I and Q symbols to be transmitted, from the data file. It must be noted that the modulation of the data takes place before and not during the actual transmission. This choice was dictated by the fact that simultaneous modulation and transmission might slow the performance of the system. The modulation process will be described analytically later.

When the modulation is over, the thread fills the channel memory buffer with the first set of data. Subsequently, it increases the *WaveRunner:: threadsReady* variable. If it is the last active channel to do so, it notifies the master thread by setting the *allChannelsReady* event.

After the above actions, the thread enters a loop, the first stage of which is a suspension of the thread until the appropriate *txBufferEmpty* event has been signaled by the interrupt service routine. The thread wakes up and checks to see if the event was caused by a buffer underflow error. In this case it simply updates

its status with the error and exits. Otherwise, it fills again the appropriate memory area with data and re-enters the suspended state.

The above loop continues as long as there is more data to transmit and the user has not cancelled transmission. When either of the two conditions occurs, the thread clears the memory buffer by writing zeroes to all the memory range. In this way, the data retrieved by the card until transmission is disabled, are simply zeroes and nothing is actually transmitted. Subsequently it updates the channel status, decreases the *WaveRunner::threadsRunning* variable and exits. If it is the last channel thread running, just before exiting, it notifies the master thread by setting the *allChannelsDone* event.

Rx thread (Figure 4-10b)**.** This thread is launched by the master thread once for every active reception channel. Its functionality is similar to the functionality of the transmission thread.

After having created the file to save the received symbols, the thread increases the *WaveRunner::threadsReady* variable and, if it is the last active thread to do so, it notifies the master thread by setting the *allThreadsReady* event.

Subsequently, it enters the reception loop. It remains suspended until notified by the appropriate *rxBufferFull* event by the interrupt service routine. When awakened, if the user has not cancelled reception and no buffer overflow error has occurred, it stores the received symbols from the buffer memory to the appropriate file and re-enters the suspended state.

When the user stops reception, an *rxBufferFull* event is set as well. The loop exits and the stored symbols are demodulated into meaningful data. The demodulation process will be described analytically later in this chapter.

Finally, the thread decreases the *WaveRunner::threadsRunning* variable, sets the *allChannelsDone* event if it is the last active channel thread and exits.

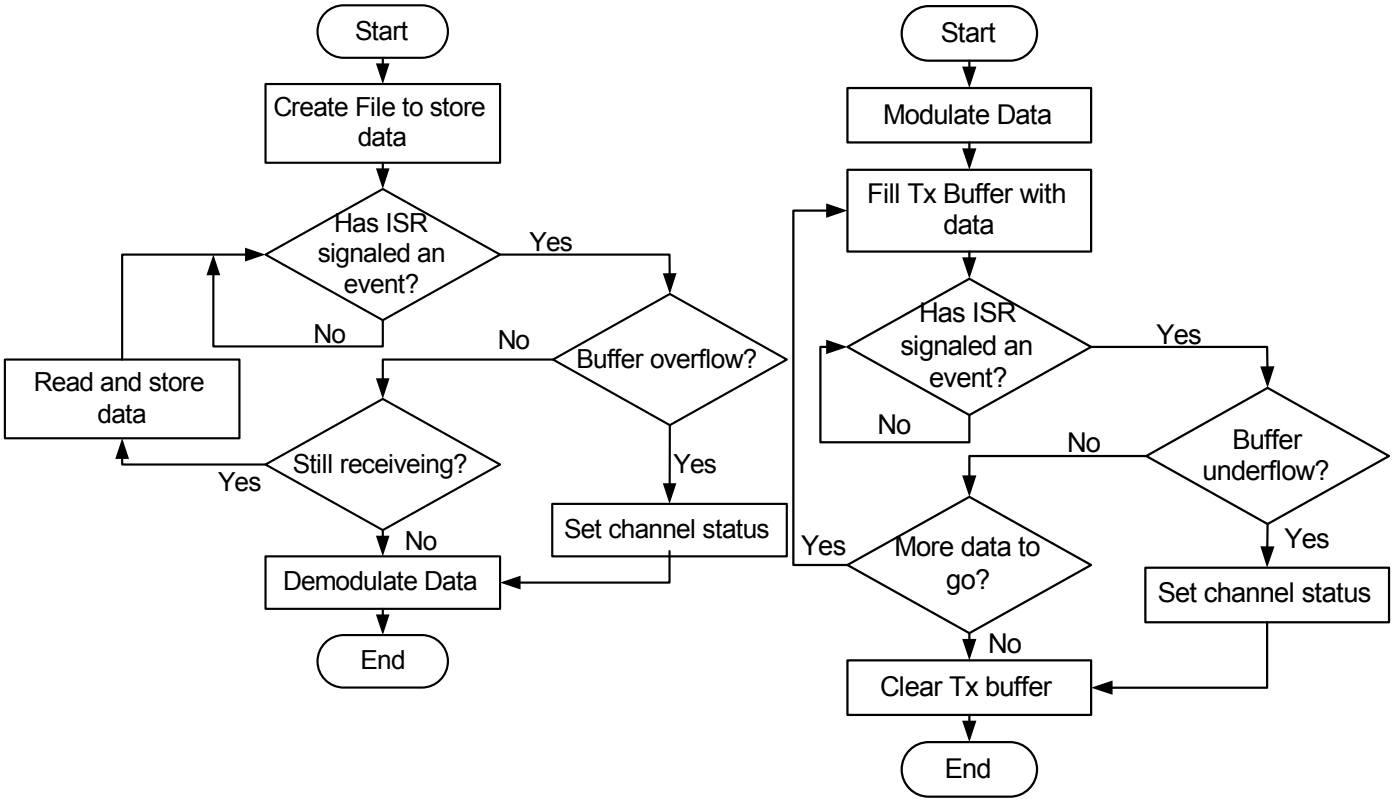


Figure 4-10. (a) Tx thread. (b) Rx thread.

F. DATA ORGANIZATION

With the procedures described so far, we have managed to achieve efficient data exchange between the card buffers and the host computer memory, as well as successful transmission and reception of symbols. Now it is time to give meaning to that data. We must organize the data in such a way that the received samples must be meaningful for the receiver. In other words, we must devise a protocol of transmission and reception.

More specifically, there are two problems which we must deal with, in order to achieve meaningful communication, phase synchronization and time synchronization.

It is extremely unlikely that the numerical oscillators of the transmitter and the receiver will be in phase. Almost certainly, they will have a random phase difference. Since the communication scheme we have chosen (M-PSK) is inherently phase coherent, this phase difference must be determined and corrected. Otherwise, in the decoding of the symbols we shall always be off by this difference.

Even after we have achieved phase synchronization, we have one more problem to solve: since in the received signal multiple samples correspond to each symbol, which is the appropriate time instance or, in other words, which of the received samples per symbol is the most appropriate one to use for the signal decoding? Here it must be noted that unlike the conventional receivers which include an integrator, our receiver consists of successive downsampling filters. Filters inherently have memory. That means that the first samples of each symbol depend not only on the current symbol but also on the previous one. So, we cannot simply average the samples. But even if we could, we should find a way to know where to start the integration, that is, at which sample each symbol starts. So, we need time synchronization.

In conventional communication systems, the above tasks are performed by dedicated carrier and clock recovery circuits, which are most of the times sophisticated and contribute to the overall complexity of the system. Moreover, most of the times they are efficient only for one communication scheme and completely inefficient for all the others. In our case, we shall try to perform the above two tasks solely by software. Here lies one of the greatest beauties of the software radio platform, which illustrates in the most profound way how this architecture contributes to the decrease of hardware complexity.

1. Transmitted Data Organization - Modulation

The symbols to be transmitted are organized in packets of 1024 symbols. Of these 1024 symbols, the first 32 are the header of the packet, while the 992 remaining symbols constitute the actual data. Of course, when transmitting the data from a file, the last packet will most likely have less than 992 symbols. This is a fact we have to take into consideration.

The composition of an individual data packet is shown at the table 4-5.

Packet Synthesis	
Symbol Position	**Significance**
1-11	Zero phase (maximum I, zero Q)
12-24	Barker Code [1 1 1 1 1 -1 -1 1 1 -1 1 -1 1]
25-32	Number of symbols in the packet
33-1024	Actual data symbols

Table 4-5. Composition of a transmitted symbols packet.

Initially we transmit 11 samples with zero phase. This can be achieved very easily by feeding the card with maximum I and zero Q samples. In doing so, we are hoping that the receiver will detect the constant phase and then correct all the subsequent packet samples phases accordingly.

After the constant phase samples, we transmit a Barker sequence of 13 symbols. The sequence is transmitted in the I channel, while the Q channel remains zero. This sequence has excellent autocorrelation properties as shown in Figure 4-11. In the receiver we are hoping that a correlator will sense the maximum of the cross correlation of the received signal with the Barker sequence and thus we shall achieve time synchronization.

The following 8 symbols after the Barker sequence denote the number of symbols per packet. As we have already stated, the number of symbols per packet may vary. So we need this information, in order to demodulate only the necessary symbols and not more. Since this piece of information is one of the most critical parts of the packet, we modulate it always in QPSK regardless of how complicated the actual data modulation may be.

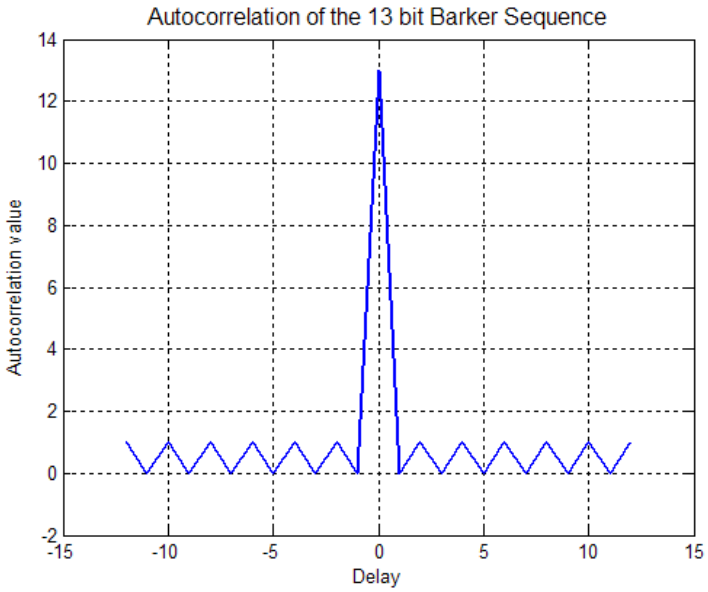


Figure 4-11. Autocorrelation properties of the 13 bit Barker code.

Finally, the actual data symbols follow, modulated by the scheme chosen by the user. In our application we restricted the possible modulation schemes to QPSK, 8-PSK and 16-PSK. It is worth mentioning that for 992 symbols per packet we need an integer number of bytes for all three schemes (more specifically, 248 bytes for the QPSK, 372 bytes for the 8-PSK and 496 bytes for the 16-PSK scheme).

We do not claim that the above packet organization was the most efficient that we could devise. It was just a simple reasonable scheme that we thought that would perform reasonably well. Of course our signal would not be transmitted in extremely noisy or distorting environments and the signal-to-noise

ratio would be very large. However, what interested us for this thesis was not to elaborate on the details but rather to simply prove that the approach works and that we could achieve with software what conventional systems need dedicated hardware to achieve.

3. Data Demodulation

The demodulation process is shown in Figure 4-12.

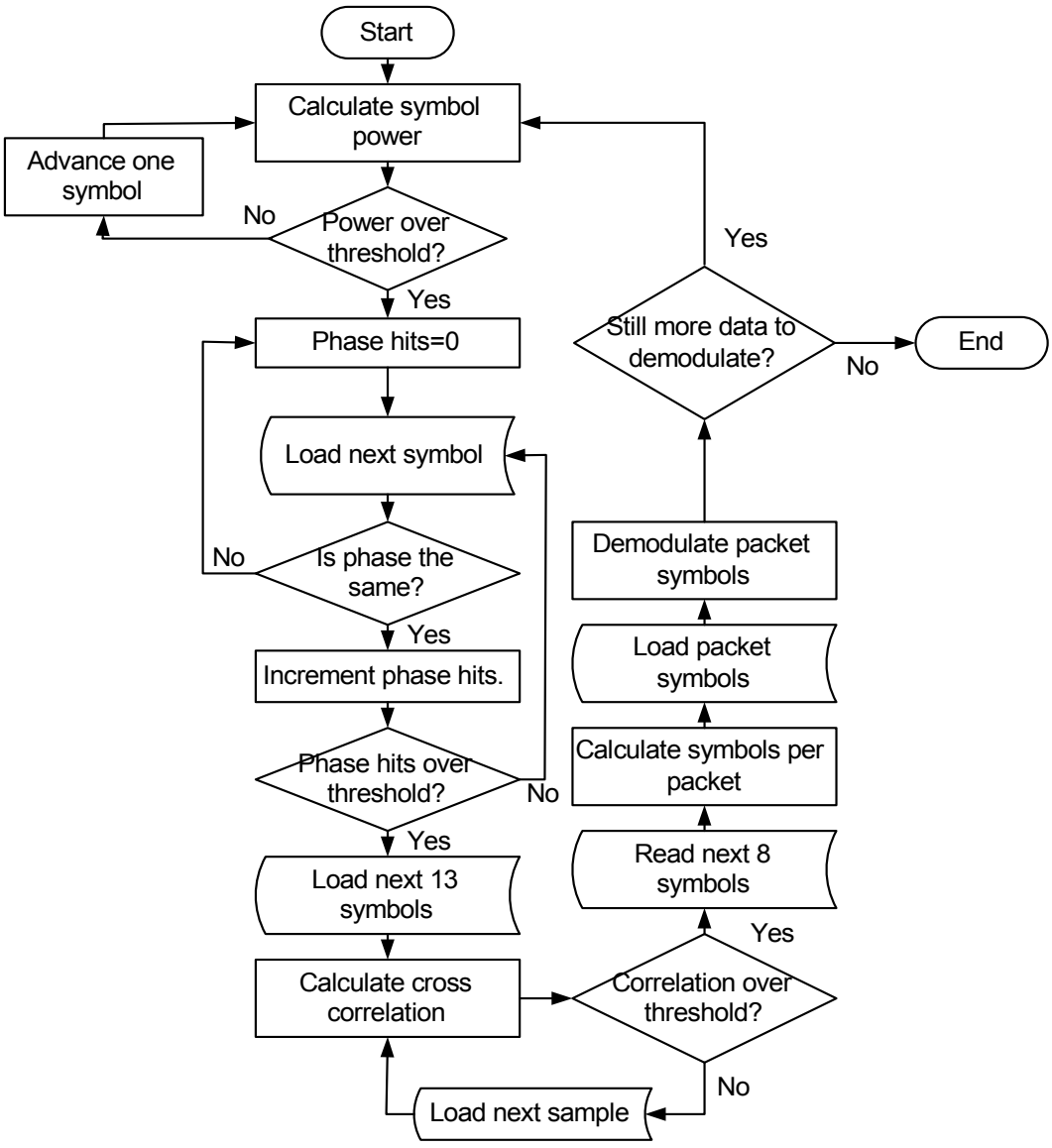


Figure 4-12. Signal demodulation process.

The procedure starts by measuring the power of every symbol and comparing it to a threshold. As long as the actual data transmission has not started yet, the power will be well below the threshold and the samples will be considered as noise and will be ignored.

As soon as the symbol power exceeds the threshold, the phase synchronization loop is started. The phase difference of successive symbols is compared against a threshold and as long as it is below the threshold, a phase hit is declared. When a predefined number of hits are achieved (six in our demonstration) at a power level above the threshold, we consider that we have detected the transmitted segment with the constant phase. The phase difference between the transmitter and the receiver numerical oscillators is simply the phase that we are measuring. All we have to do is to subtract this phase from the phase of the subsequent symbols and we are done. We have achieved phase synchronization.

Following the successful synchronization in phase, we slide the received symbols in a software-implemented correlator, which computes the cross correlation between the received signal and the Barker code. The moment the cross correlation exceeds a threshold (12 in our case), we consider that we are positioned in the beginning of the Barker code in the received signal. So, we have achieved time synchronization.

When time synchronization has been achieved, we simply advance by 13 symbols and read the next 8 symbols, demodulate them using a QPSK demodulation scheme and calculate the number of symbols per packet.

What remains to be done, is to successively read and demodulate the actual data symbols of the packet. After this task has been completed, the whole procedure is repeated from the beginning for each packet, until the data file runs out of data.

In the above effort to synchronize, we have used several thresholds: a power threshold, a maximum phase difference threshold, a phase hits threshold and a cross correlation threshold. The choices of the values of these thresholds were not based on any sophisticated algorithms; we simply chose values that seemed reasonable to us. However, it is well understood that in a real environment where a large signal-to-noise ratio may not be achievable, the setting of the proper values for these thresholds may become much more complicated. There may even exist instances that our procedure may not work at all. However, as we have already mentioned, at this stage we were not so much interested in elaborating the details but rather in proving that our approach works. So we kept the detection algorithm relatively simple.

G. CHOICE OF THE PROPER FILTERS

Although the choice of the appropriate filters for the transmitter interpolation and the receiver decimation stages may seem a trivial task to perform, it is actually one of the most critical factors for the overall performance of the system. If the filters are chosen inappropriately, intersymbol interference may cause errors in the symbol detection even under large signal-to-noise ratios.

For the needs of our application, we chose a square root raised cosine filter for the transmitter with a cutoff frequency of π/4, an interpolation factor of 4 and a roll-off factor varying from 0.25 to 0.10. The frequency response of the filter for a roll-off factor of 0.15 is shown in Figure 4-13. As we can see, the filter does not cancel signal aliasing completely (its stopband does not start exactly at π/D), but rather permits it in a controllable way, so that it will be canceled by the filters of the receiver.

In the receiver, we used a decimation of 16, performed in two stages of 4. At each stage the signal was filtered by a low pass filter, whose frequency response is shown in Figure 4-14.

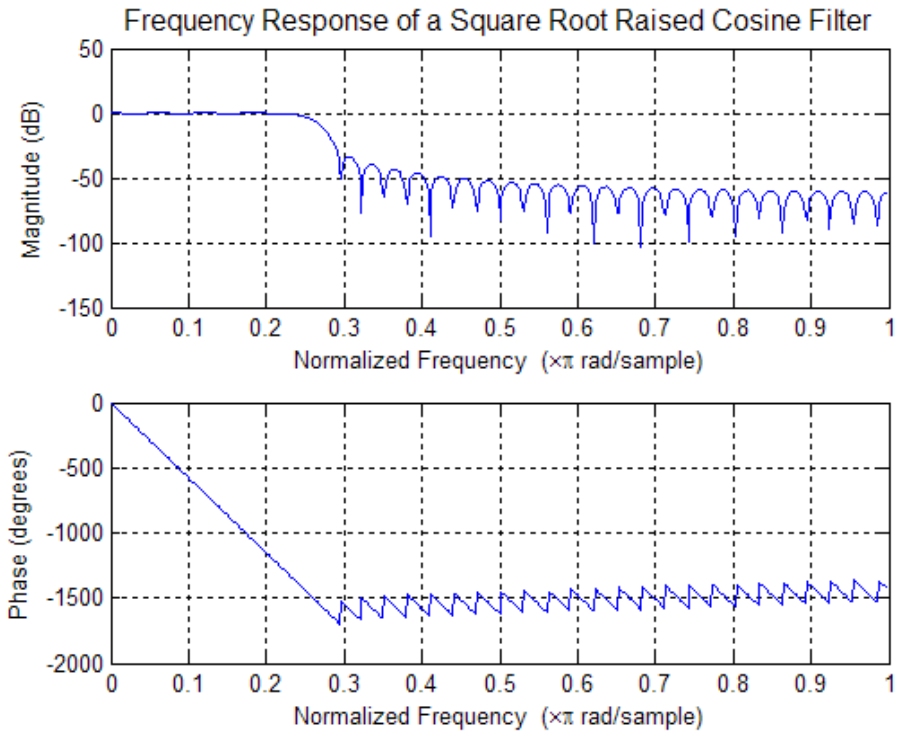


Figure 4-13. Frequency response of the transmitter shaping filter.

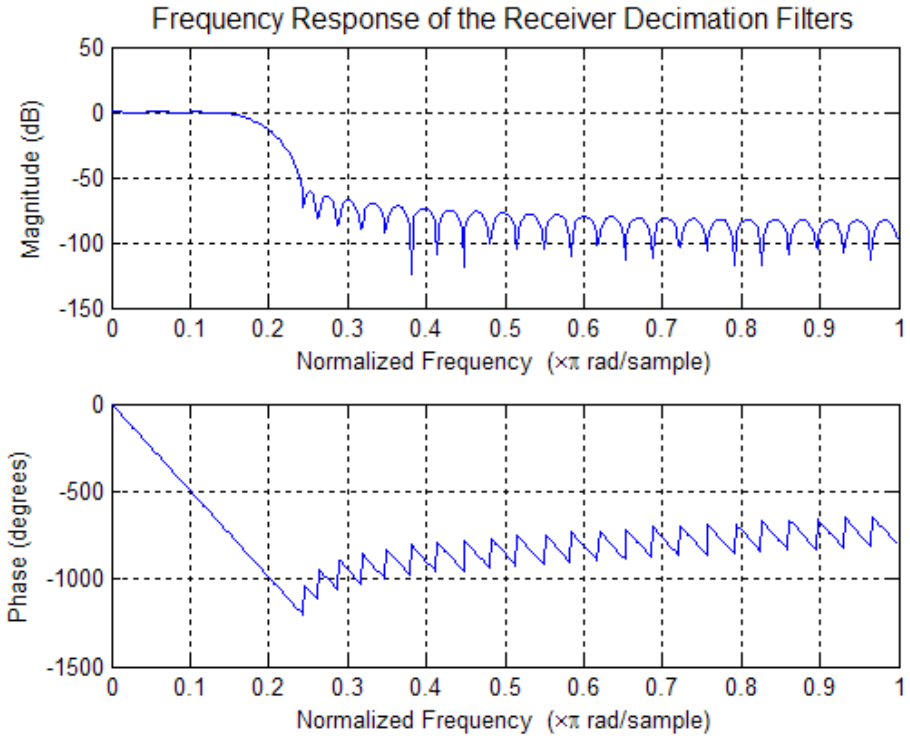


Figure 4-14. Frequency response of the receiver decimation filters.

H. FROM HERE …

In the current chapter we outlined the main aspects of our work and the functionality of the code we wrote. Now it is time to see the results of the tests we ran with this code. Did it really work and what results did we get? This is the subject of the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

V: RESULTS - CONCLUSIONS

A. INTRODUCTION

In the previous chapter we described analytically the application that we developed and the procedures that are involved in the implementation of the communication link, both at the hardware management level as well as at the organization of data in a meaningful protocol.

Now it is the time to see the results of this effort. This chapter will not present any diagrams and curves, because this was not the nature of our work. Our result analysis will consist mainly of the presentation of some oscilloscope snapshots in order to let the reader witness what we saw at the lab and an insight into the streams of transmitted and received data, in order to prove and explain that our algorithm worked successfully.

B. TEST BENCH

In order to test and run our application, we set up a test bench in the microwave lab of the Spanagel building, as shown in Figure 5-1. The transceivers were hosted into two PC-type personal computers having the following features:

- Intel Pentium 4 processors at 3.05 GHz with multithreaded technlology
- 512 Mbytes of RAM at 1024 MHz (So that the RAM speed could closely follow the speed of the processor bus and would cause no less bottlenecks).
- Two SCSI hard disks with 18 GBytes capacity each (The choice of the SCSI protocol was imperative, because we wanted to ensure that data transfers between the computer memory and the hard disks would

occur as fast as possible – certainly at a rate compared to the rate of data transfer between the transceiver and the host computer).

Figure 5-1. Test bench used for the tests of the code.

For each host computer we set up a line of measuring instruments, consisting of:

- A Tektronix 475 Signal Oscilloscope
- A Tektronix 492 Spectrum Analyzer
- A HP 436A Power meter.
- A Wavetek 148 signal generator.

By the end of our experiments, we used the Tektronix TDS 3012B Digital Oscilloscope – Spectrum Analyzer, in order to capture the oscilloscope images shown at the next paragraph.

C. OSCILLOSCOPE IMAGES

Figures 5-2 to 5-7 show some snapshots of the images of the oscilloscope. Th upper part of each figure (blue curve) is the signal in the time domain, while the lower part (red curve) is the signal in the frequency domain.

Initially we generated some simple tones (signal without modulation). This type of signal can be presented very easily at the oscilloscope screen, because it is very easy to synchronize. Especially in Figure 5-4, we demonstrate the ease of creating eight tones (one per transmission channel). At the same time eight recepton channels may be active in order to receive the signal.

Figures 5-5 to 5-7 show some actual QPSK modulated channels. With a careful observation of the images, the different phases of the signal can be easily distinguished. A more thorough analysis of the received waveforms will follow in the next paragraph.

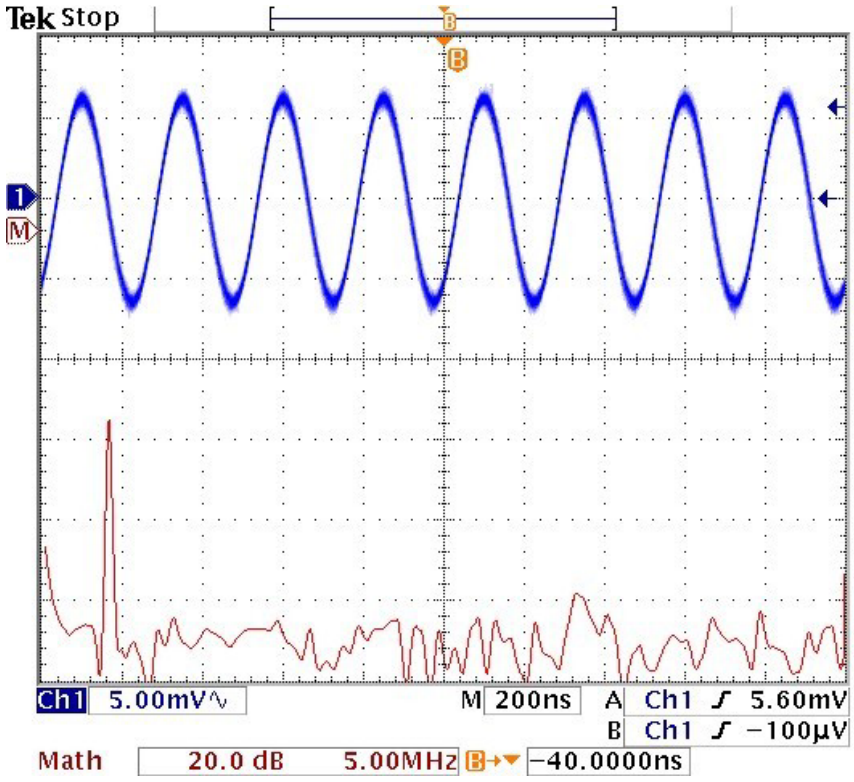


Figure 5-2. One tone at 4 MHz.

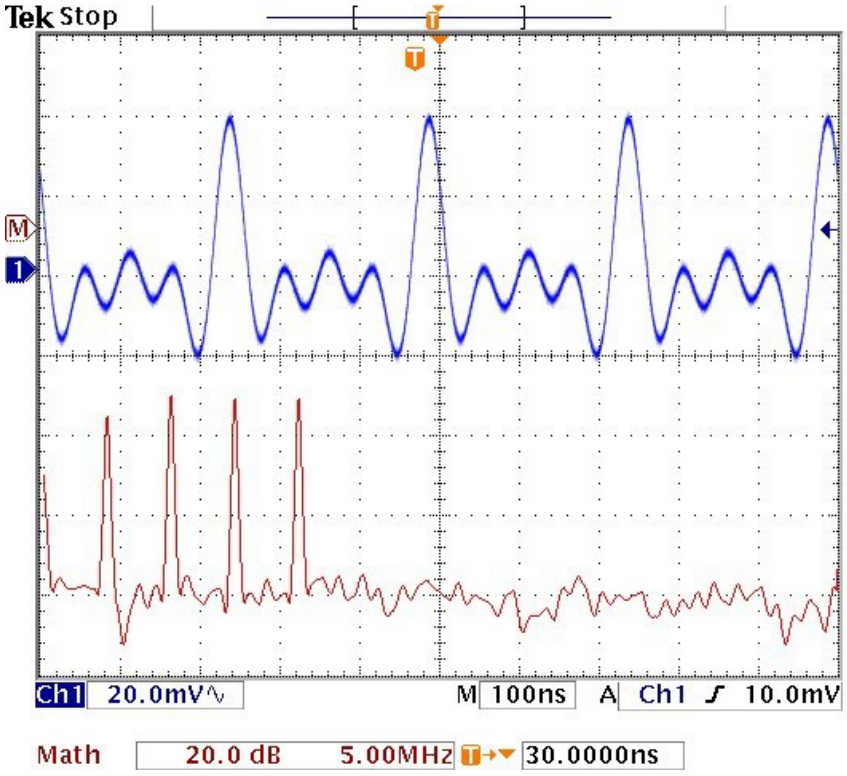


Figure 5-3. Four tones at 4, 8, 12 and 16 MHz.

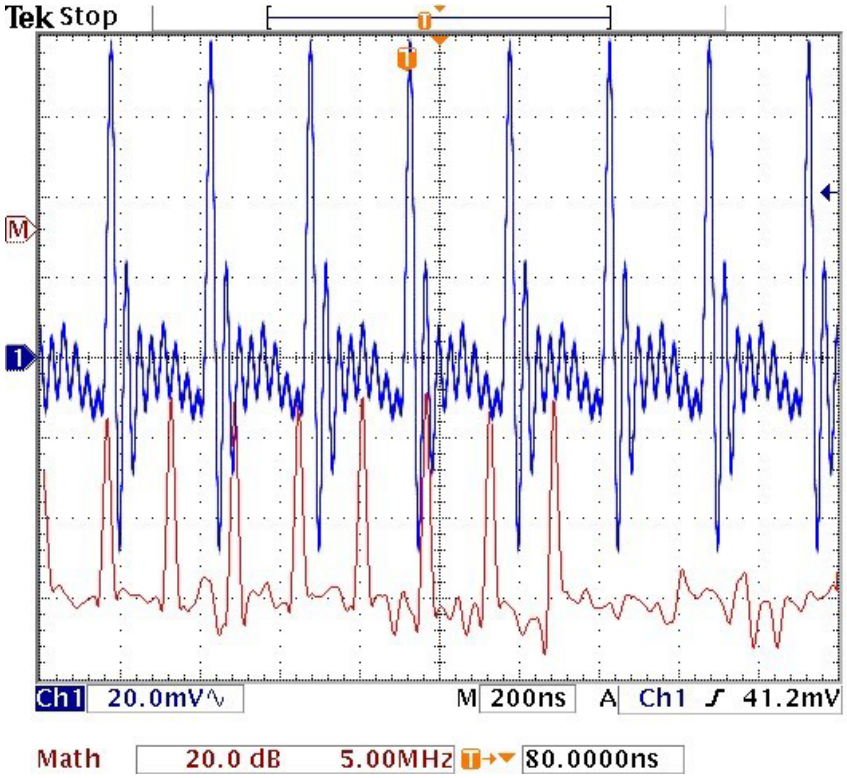


Figure 5-4. Eight tones at 4,8,12,16,20,24,28,32 and 36 MHz.

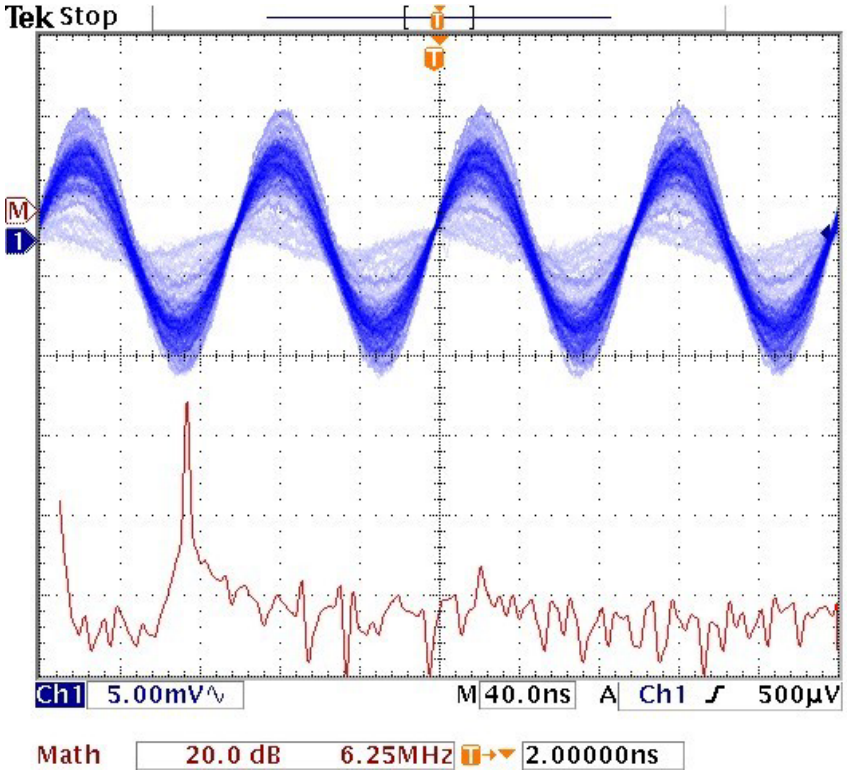


Figure 5-5. One QPSK channel at 5 MHz.

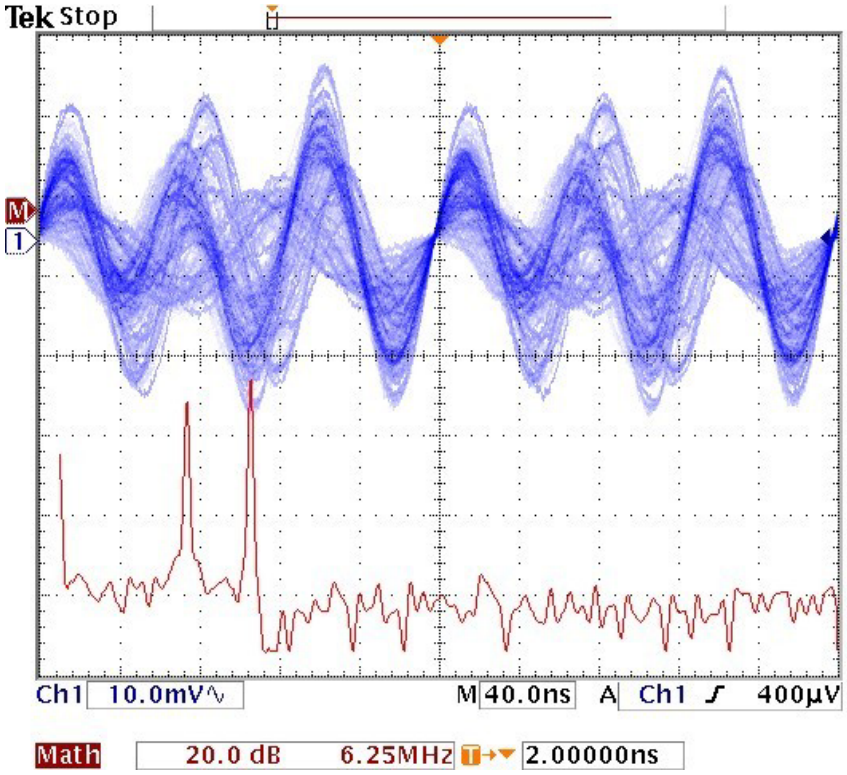


Figure 5-6. Two QPSK channels at 10 and 15 MHz.

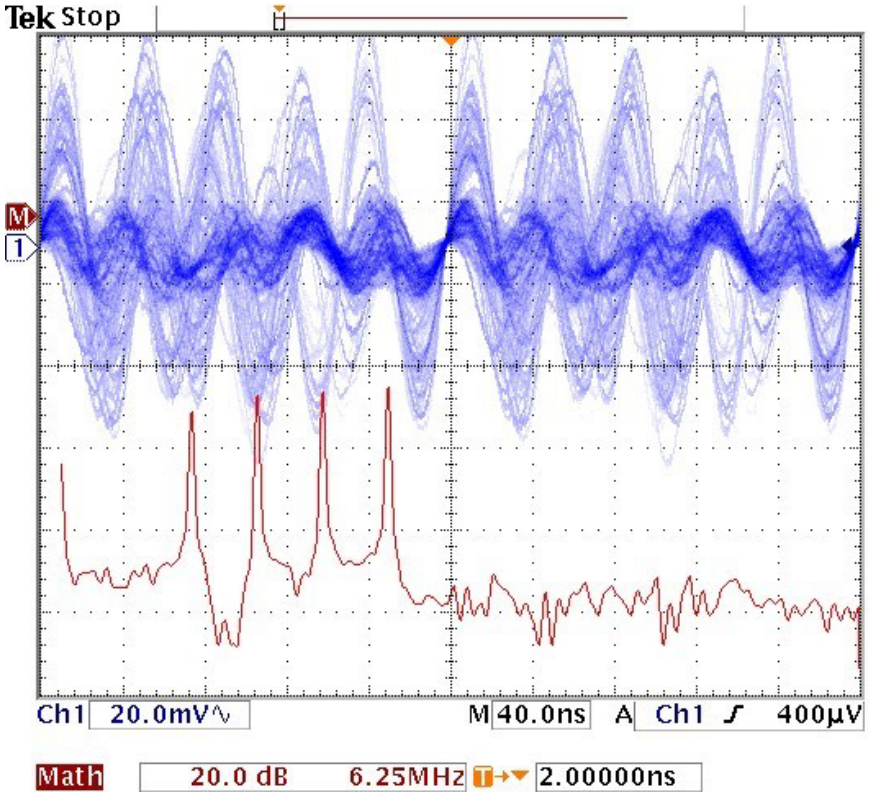


Figure 5-7. Four QPSK channels at 10, 15, 20 and 25 MHz.

D. DATA WAVEFORMS ANALYSIS

Figure 5-8 shows the first 100 symbols of a transmitted data packet. A careful observation of the figure reveals the structure of the packet. As we can see, the symbols have a constant envelope, since they are BPSK (the Barker code) or QPSK (the number of symbols per packet) modulated. Also, from the signal phase diagram, the segments intended for the phase synchronization and the Barker code segment are very easily distinguishable.

Figure 5-9 shows the received waveform before phase synchronization. As we can easily see, the segment with the steady phase is very easily recognized. After our code senses this phase and compensates for it, the waveform that results is shown in Figure 5-10. We can easily see two things, the phase of the steady-phase segment is now zero and the Barker code is very

easily distinguishable. The received waveform now is clearly our transmitted waveform shaped by the shaping filters.

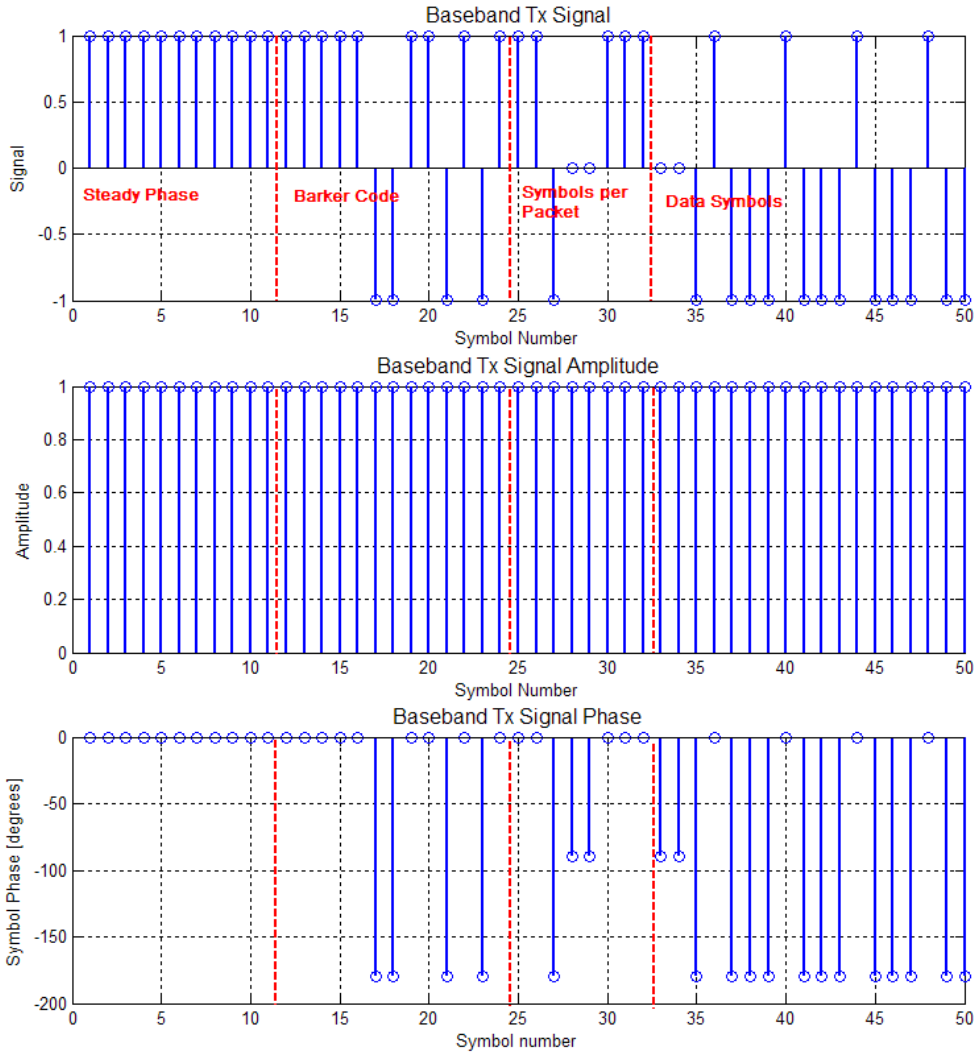


Figure 5-8. Transmitted baseband waveform.

Figure 5-11 shows the cross correlation between the waveform of Figure 5-10 and the Barker code. As we can see, at the point of synchronization the normalized cross-correlation value (cross correlation divided by the signal power) is very close to its theoretical maximum of 13 (in our case it is 12.5). Also, nowhere near this value does the cross correlation even approach this peak. It is

very easy for the code to sense this peak and declare time synchronization at this point. Of course, as the Figure 5-11 demonstrates, there may exist large cross correlation values before the beginning of reception of the actual signal, due to the random phase of the received signal. However, at that segment the received power is well below the threshold. So we simply ignore these values.

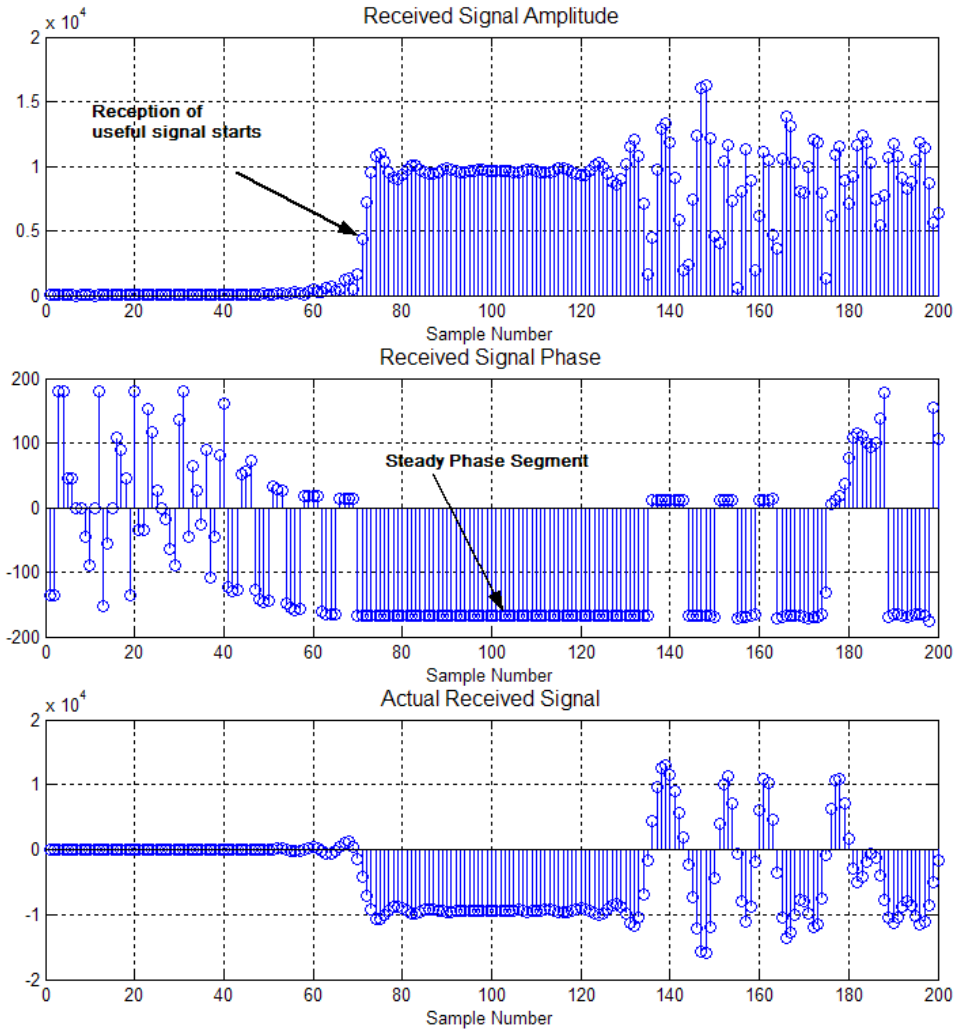


Figure 5-9. Received Baseband Signal Waveform, before phase difference compensation.

E. TESTS RESULTS

Using the set up described in section B of this chapter, we tried all the possible modulation schemes for which the code was developed, from simple test tones to 16-PSK schemes.

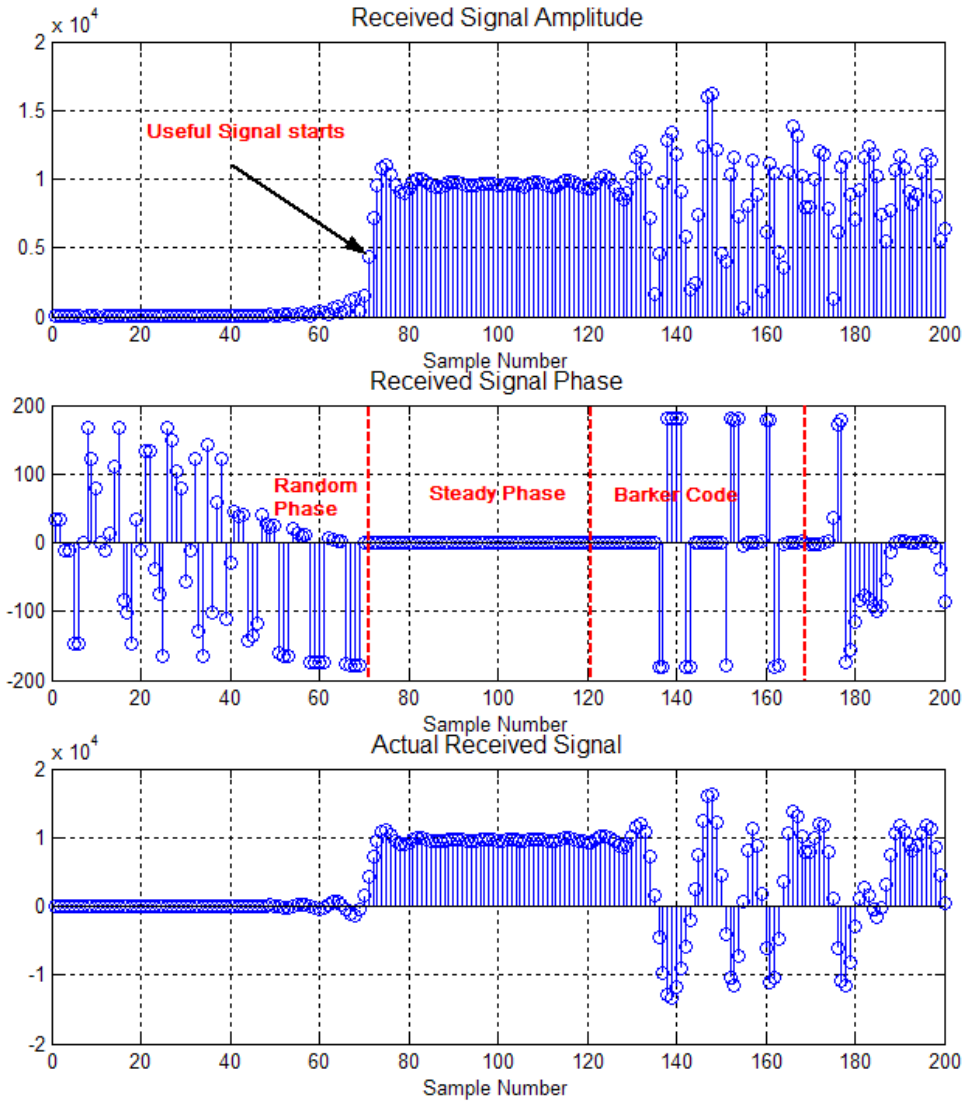


Figure 5-10. Received Baseband Signal Waveform, after phase difference compensation.

At the transmitter the signal was fed to the card at a datarate of 58,125 symbols per second. There it was shaped by the square root raised-cosine filter described at the previous chapter, interpolated by a factor of 4, transformed into analog format and transmitted.

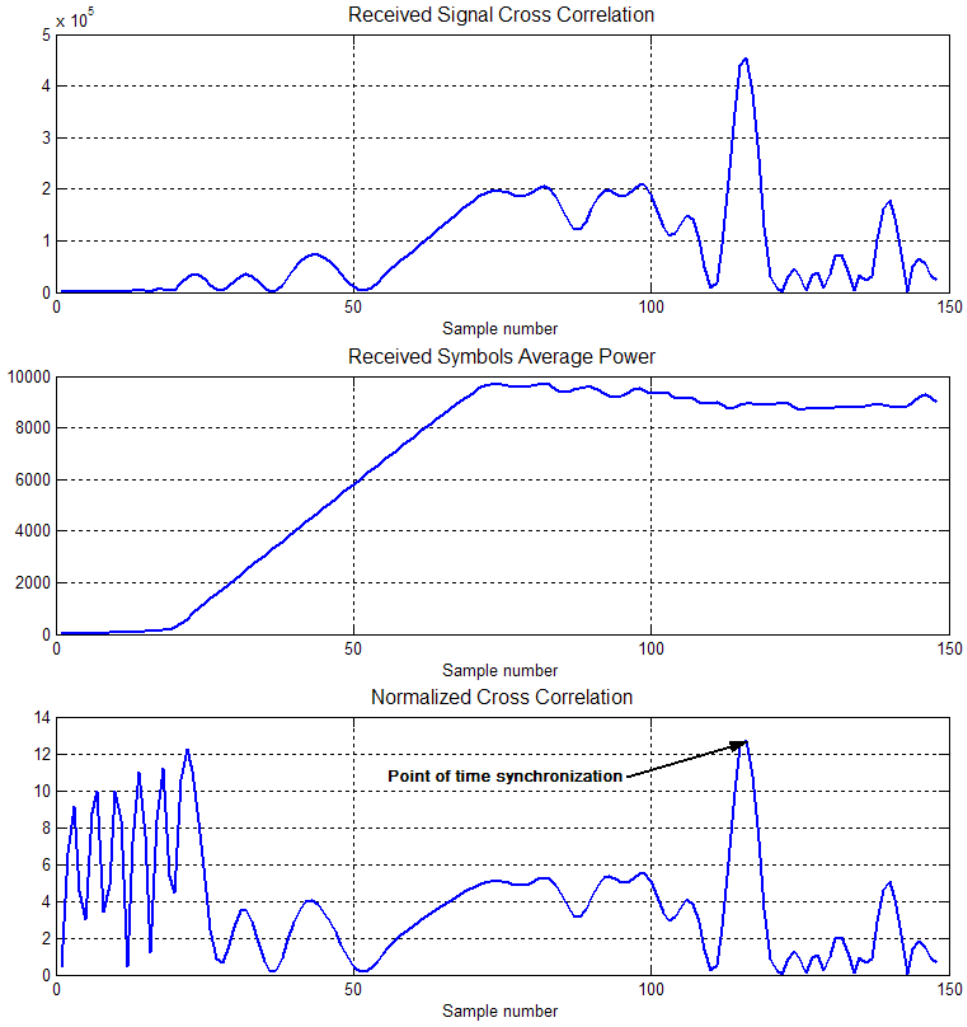


Figure 5-11. Cross correlation between the received baseband waveform after phase correction, and the Barker code.

At the receiver, the signal was initially decimated by the CIC filter by a factor of 25. This decimation rendered a data throughput of the CIC filter of 3.72 Mbps. Subsequently, the signal was fed into the FIR engine of the DDC and was

further decimated by 16 in two stages of 4, using the shaping LPF described also at the previous chapter.

This procedure gave us a final output rate of 232.5 ksps, which means 4 samples per transmitted symbol. Taking into account the fact that the null-to-null bandwidth of an M-PSK signal is twice the symbol rate, the bandwidth of our received signal included the main lobe and the first sidelobe of the transmitted signal, that is roughly 95% of the transmitted signal power. We might even be able to reduce the output rate to two samples per symbol by further decimating the received signal by a factor of 2, but that would need more carefully designed shaping filters and we might not be able to mitigate the intersymbol interference completely. So we chose the rate of 4 samples per symbol.

All the simulations gave very good results! Phase and time synchronization was achieved without problems. The received signal was demodulated and the transmitted waveform was restored successfully. Even the last packet of the file, where the number of symbols was less than 992 and we could not evade rounding effects, was demodulated without problem. So, the above results confirmed the correctness of our code.

The last issue we have to mention concerns the maximum achievable datarate of the system. In order to measure it, we activated several channel combinations from one Tx channel only to all 8 channel pairs working. (Of course, in this case we were not interested in the correct demodulation of the received signal, since we had not configured the channel filters properly. We only wanted to see what would be the maximum datarates that we could achieve).

In every case, we achieved a total datarate between 4.2 Msps and 5.5 Msps before a buffer overflow or underflow occurred. This speed was achieved using the 32-bit PCI bus of our host computers. The transceivers can also accommodate 64-bit transfers. The manufacturer claims that the maximum

datarate can reach easily 8 Msps. It is quite possible that had we used more dedicated hosts with 64-bit PCI buses, we would have achieved these rates.

VI: FIELDS FOR FURTHER STUDY

Our thesis demonstrated in a profound way some of the benefits and the potentials of the Software Defined Radio technology. Complicated functionality, that until recently required dedicated hardware to be implemented, can now be very easily implemented solely by software. The communications transceiver has become more lightweight and generic, while the software has given it the ability to adapt to a variety of standards and schemes. But perhaps the most important of all its benefits is the fact the cost of the hardware has been reduced dramatically. This fact combined with the progress in computer hardware, have rendered it possible to set up and implement communication systems at a fraction of the cost we would need until recently.

We do not claim that with this thesis we have exhausted the subject. The technology is so new and its capabilities are so enormous that we only scratched the tip of the iceberg. The possibilities for further development using the hardware we used in our thesis are endless. Just to name some:

- Connection of the hardware to a wireless RF frontend, in order to achieve wireless communication.
- Addition of a programmable RF frequency upconverter in order to be able to sweep several frequency bands.
- Addition of error coding to our signal in order to reduce further the probability of error.
- Study of the system performance in a variety of noise environments and signal-to-noise ratios.
- Effort to implement a known communications protocol using the hardware.
- Frequency spectrum measurement and analysis.

- Adaptive use of the available spectrum.

The last one of the above fields of study is probably one of the most interesting application areas of the software radio technology. Since the system can be used easily for both communication and spectrum measurements, the communicating parts could implement a protocol in order to analyze the noise profile of the available spectrum and use it optimally (in other words direct their power to the least occupied portions of the spectrum). In fact, in another thesis developed in parallel with this one (Ref. 10), Captain Nikos Apostolou has established a theoretical background on how this could be achieved. We were hoping to be able to implement this theory in practice using the hardware, but unfortunately, due to time restrictions, we did not manage to do so. This would be probably the most interesting field of study for another student to continue our work.

As a closing statement of this text, the author like to state once more how exciting the work on this thesis was. Despite the endless hours of frustration in the lab, when nothing seemed to be working properly, when the good results finally showed up, our satisfaction was unparalleled. The ability to control a sophisticated piece of hardware using a common programming language was something that has always fascinated the author. Finally, this thesis gave the author the opportunity to apply in practice a lot of the theory he had learnt in previous classes, which was a very valuable experience as well.

APPENDIX A: CODE LISTING

Table A-1 shows the files of the *WaveRadio* project and their description. The listing of the code of these files follows in the next pages.

File Name	Description
WaveRadio.h WaveRadio.cpp	Main application classes and the application initialization code.
MainFrm.h MainFrm.cpp ChildFrm.h ChildFrm.cpp ChildView.h ChildView.cpp	Main and child windows classes and manipulating functions.
CommsCtrlDlg.h CommsCtrlDlg.cpp	The communications panel main class, hosting the three panel pages.
CommsTab1.h CommsTab1.cpp CommsTab2.h CommsTab2.cpp CommsTab3.h CommsTab3.cpp	Classes encapsulating the functionality of the communication panel pages.
GlobalVars.h	Header file containing global variables that need to be accessed by all the files of the application.
WaveRunner.h WaveRunner.cpp	Class containing the functionality of the one and only WaveRunner object of the application.
WaveRunnerChannel.h WaveRunnerChannel.cpp	Abstract class, used as a building block for the transmission and reception channels.
RxChannel.h RxChannel.cpp	Class containing the functionality of the reception channels.
TxChannel.h TxChannel.cpp	Class containing the functionality of the transmission channels.
WaveRunnerIsr.cpp	It contains the Interrupt Service Routine, the master Rx/Tx thread and the individual Rx and Tx threads.
Modemod.cpp	Contains the modulation and demodulation routines.
Memory_map.h	Maps the WaveRunner registers addresses to constants, for easier use by the application files.
Pmcradioi.h	Contains the function prototypes of the WaveRunner library.

Table A-1: WaveRadio project files description.

The above listing does not include the resource files of the project, as well as the library file `pmcradio.lib` which contains the actual code of the WaveRunner library functions and must be included in the project, in order for the code to compile successfully.

WAVERADIO.H

```
// WaveRadio.h : main header file for the WaveRadio application
//
#pragma once

#ifndef __AFXWIN_H__
	#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"		// main symbols

// CWaveRadioApp:
// See WaveRadio.cpp for the implementation of this class
//

class CWaveRadioApp : public CWinApp
{
public:
	CWaveRadioApp();


// Overrides
public:
	virtual BOOL InitInstance();
	virtual int ExitInstance();

// Implementation
protected:
	HMENU m_hMDIMenu;
	HACCEL m_hMDIAccel;

public:
	afx_msg void OnAppAbout();
	afx_msg void OnFileNew();
	DECLARE_MESSAGE_MAP()
	afx_msg void OnApplicationsCommspanel();
};

extern CWaveRadioApp theApp;
```

WAVERADIO.CPP

```
// WaveRadio.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
```

```
#include "WaveRadio.h"
#include "MainFrm.h"

#include "ChildFrm.h"
#include "CommsCtrlDlg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#endif

#include "WaveRunner.h"

// CWaveRadioApp

BEGIN_MESSAGE_MAP(CWaveRadioApp, CWinApp)
	ON_COMMAND(ID_HELP_ABOUT, OnAppAbout)
	ON_COMMAND(ID_FILE_NEW, OnFileNew)
	ON_COMMAND(ID_APPLICATIONS_COMMSPANEL, OnApplicationsCommspanel)
END_MESSAGE_MAP()


// CWaveRadioApp construction

CWaveRadioApp::CWaveRadioApp()
{
	// TODO: add construction code here,
	// Place all significant initialization in InitInstance
}


// The one and only CWaveRadioApp object

CWaveRadioApp theApp;
WaveRunner* wr=WaveRunner::getNewWaveRunner();

// CWaveRadioApp initialization

BOOL CWaveRadioApp::InitInstance()
{
	int error=wr->Open();
	if (error)
	{
		CString disp;
		disp.Format("Error: %2d\nWaveRunner card could not be
opened.\n Process will abort.",error);
		AfxMessageBox(disp,MB_OK,0);
		//return FALSE;
	}
	// InitCommonControls() is required on Windows XP if an
application
	// manifest specifies use of ComCtl32.dll version 6 or later to
enable
	// visual styles.  Otherwise, any window creation will fail.
	InitCommonControls();

	CWinApp::InitInstance();

	// Initialize OLE libraries
```

```
	if (!AfxOleInit())
	{
		AfxMessageBox(IDP_OLE_INIT_FAILED);
		return FALSE;
	}
	AfxEnableControlContainer();
	// Standard initialization
	// If you are not using these features and wish to reduce the
size
	// of your final executable, you should remove from the following
	// the specific initialization routines you do not need
	// Change the registry key under which our settings are stored
	// TODO: You should modify this string to be something
appropriate
	// such as the name of your company or organization
	SetRegistryKey(_T("Local AppWizard-Generated Applications"));
	// To create the main window, this code creates a new frame
window
	// object and then sets it as the application's main window
object
	CMDIFrameWnd* pFrame = new CMainFrame;
	m_pMainWnd = pFrame;
	// create main MDI frame window
	if (!pFrame->LoadFrame(IDR_MAINFRAME))
		return FALSE;
	// try to load shared MDI menus and accelerator table
	//TODO: add additional member variables and load calls for
	//	additional menu types your application may need
	HINSTANCE hInst = AfxGetResourceHandle();
	m_hMDIMenu  = ::LoadMenu(hInst,
MAKEINTRESOURCE(IDR_WaveRadioTYPE));
	m_hMDIAccel = ::LoadAccelerators(hInst,
MAKEINTRESOURCE(IDR_WaveRadioTYPE));
	// The main window has been initialized, so show and update it
	pFrame->ShowWindow(m_nCmdShow);
	pFrame->UpdateWindow();
	return TRUE;
}


// CWaveRadioApp message handlers

int CWaveRadioApp::ExitInstance()
{
	//TODO: handle additional resources you may have added
	if(wr->cardStatus==0)
	{
		int error=wr->Close();
		if (error)
		{
			CString disp;
			disp.Format("Error: %2d\nWaveRunner card could not be
closed.",error);
			AfxMessageBox(disp,MB_OK,0);
		}
	}
```

```
	delete wr;
	if (m_hMDIMenu != NULL)
		FreeResource(m_hMDIMenu);
	if (m_hMDIAccel != NULL)
		FreeResource(m_hMDIAccel);

	return CWinApp::ExitInstance();
}


void CWaveRadioApp::OnFileNew()
{
	CMainFrame* pFrame = STATIC_DOWNCAST(CMainFrame, m_pMainWnd);
	// create a new MDI child window
	pFrame->CreateNewChild(
		RUNTIME_CLASS(CChildFrame), IDR_WaveRadioTYPE, m_hMDIMenu,
m_hMDIAccel);
}


// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
	CAboutDlg();

// Dialog Data
	enum { IDD = IDD_ABOUTBOX };

protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support

// Implementation
protected:
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
END_MESSAGE_MAP()

// App command to run the dialog
void CWaveRadioApp::OnAppAbout()
{
	CAboutDlg aboutDlg;
	aboutDlg.DoModal();
}
```

```
// CWaveRadioApp message handlers


void CWaveRadioApp::OnApplicationsCommspanel()
{
	// TODO: Add your command handler code here
	CCommsCtrlDlg commsCtrlDlg("Main Comms Control Panel");
	commsCtrlDlg.DoModal();
}
```

MAINFRM.H

```
// MainFrm.h : interface of the CMainFrame class
//


#pragma once
class CMainFrame : public CMDIFrameWnd
{
	DECLARE_DYNAMIC(CMainFrame)
public:
	CMainFrame();

// Attributes
public:

// Operations
public:

// Overrides
public:
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Implementation
public:
	virtual ~CMainFrame();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:  // control bar embedded members
	CStatusBar  m_wndStatusBar;
	CToolBar    m_wndToolBar;

// Generated message map functions
protected:
	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
	DECLARE_MESSAGE_MAP()
};
```

MAINFRM.CPP

```
// MainFrm.cpp : implementation of the CMainFrame class
```

```
//

#include "stdafx.h"
#include "WaveRadio.h"

#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CMainFrame

IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
      ON_WM_CREATE()
END_MESSAGE_MAP()

static UINT indicators[] =
{
      ID_SEPARATOR,           // status line indicator
      ID_INDICATOR_CAPS,
      ID_INDICATOR_NUM,
      ID_INDICATOR_SCRL,
};


// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
      // TODO: add member initialization code here
}

CMainFrame::~CMainFrame()
{
}


int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
      if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
            return -1;

      if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
WS_VISIBLE | CBRS_TOP
            | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC) ||
            !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
      {
            TRACE0("Failed to create toolbar\n");
            return -1;      // fail to create
      }

      if (!m_wndStatusBar.Create(this) ||
```

```
            !m_wndStatusBar.SetIndicators(indicators,
              sizeof(indicators)/sizeof(UINT)))
      {
            TRACE0("Failed to create status bar\n");
            return -1;      // fail to create
      }
      // TODO: Delete these three lines if you don't want the toolbar
to be dockable
      m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
      EnableDocking(CBRS_ALIGN_ANY);
      DockControlBar(&m_wndToolBar);

      return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
      if( !CMDIFrameWnd::PreCreateWindow(cs) )
            return FALSE;
      // TODO: Modify the Window class or styles here by modifying
      //  the CREATESTRUCT cs

      return TRUE;
}


// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
      CMDIFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
      CMDIFrameWnd::Dump(dc);
}

#endif //_DEBUG


// CMainFrame message handlers
```

CHILDFRM.H

```
// ChildFrm.h : interface of the CChildFrame class
//


#pragma once

#include "ChildView.h"

class CChildFrame : public CMDIChildWnd
{
      DECLARE_DYNCREATE(CChildFrame)
```

```
public:
	CChildFrame();

// Attributes
public:

// Operations
public:

// Overrides
	public:
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
	virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra,
AFX_CMDHANDLERINFO* pHandlerInfo);

// Implementation
public:
	// view for the client area of the frame.
	CChildView m_wndView;
	virtual ~CChildFrame();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

// Generated message map functions
protected:
	afx_msg void OnFileClose();
	afx_msg void OnSetFocus(CWnd* pOldWnd);
	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
	DECLARE_MESSAGE_MAP()
};
```

CHILDFRM.CPP

```
// ChildFrm.cpp : implementation of the CChildFrame class
//
#include "stdafx.h"
#include "WaveRadio.h"

#include "ChildFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CChildFrame

IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)

BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
	ON_COMMAND(ID_FILE_CLOSE, OnFileClose)
	ON_WM_SETFOCUS()
	ON_WM_CREATE()
END_MESSAGE_MAP()
```

```
// CChildFrame construction/destruction

CChildFrame::CChildFrame()
{
	// TODO: add member initialization code here
}

CChildFrame::~CChildFrame()
{
}


BOOL CChildFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: Modify the Window class or styles here by modifying the
CREATESTRUCT cs
	if( !CMDIChildWnd::PreCreateWindow(cs) )
		return FALSE;

	cs.dwExStyle &= ~WS_EX_CLIENTEDGE;
	cs.lpszClass = AfxRegisterWndClass(0);
	return TRUE;
}


// CChildFrame diagnostics

#ifdef _DEBUG
void CChildFrame::AssertValid() const
{
	CMDIChildWnd::AssertValid();
}

void CChildFrame::Dump(CDumpContext& dc) const
{
	CMDIChildWnd::Dump(dc);
}

#endif //_DEBUG


// CChildFrame message handlers
void CChildFrame::OnFileClose()
{
	// To close the frame, just send a WM_CLOSE, which is the
equivalent
	// choosing close from the system menu.
	SendMessage(WM_CLOSE);
}

int CChildFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CMDIChildWnd::OnCreate(lpCreateStruct) == -1)
		return -1;

	// create a view to occupy the client area of the frame
```

```
	if (!m_wndView.Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
		CRect(0, 0, 0, 0), this, AFX_IDW_PANE_FIRST, NULL))
	{
		TRACE0("Failed to create view window\n");
		return -1;
	}

	return 0;
}

void CChildFrame::OnSetFocus(CWnd* pOldWnd)
{
	CMDIChildWnd::OnSetFocus(pOldWnd);

	m_wndView.SetFocus();
}

BOOL CChildFrame::OnCmdMsg(UINT nID, int nCode, void* pExtra,
AFX_CMDHANDLERINFO* pHandlerInfo)
{
	// let the view have first crack at the command
	if (m_wndView.OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
		return TRUE;

	// otherwise, do default handling
	return CMDIChildWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
}
```

CHILDVIEW.H

```
// ChildView.h : interface of the CChildView class
//


#pragma once


// CChildView window

class CChildView : public CWnd
{
// Construction
public:
	CChildView();

// Attributes
public:

// Operations
public:

// Overrides
	protected:
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Implementation
public:
```

```
	virtual ~CChildView();

	// Generated message map functions
protected:
	afx_msg void OnPaint();
	DECLARE_MESSAGE_MAP()
};
```

CHILDVIEW.CPP

```
// ChildView.cpp : implementation of the CChildView class
//

#include "stdafx.h"
#include "WaveRadio.h"
#include "ChildView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CChildView

CChildView::CChildView()
{
}

CChildView::~CChildView()
{
}


BEGIN_MESSAGE_MAP(CChildView, CWnd)
	ON_WM_PAINT()
END_MESSAGE_MAP()



// CChildView message handlers

BOOL CChildView::PreCreateWindow(CREATESTRUCT& cs)
{
	if (!CWnd::PreCreateWindow(cs))
		return FALSE;

	cs.dwExStyle |= WS_EX_CLIENTEDGE;
	cs.style &= ~WS_BORDER;
	cs.lpszClass =
AfxRegisterWndClass(CS_HREDRAW|CS_VREDRAW|CS_DBLCLKS,
		::LoadCursor(NULL, IDC_ARROW),
reinterpret_cast<HBRUSH>(COLOR_WINDOW+1), NULL);

	return TRUE;
}

void CChildView::OnPaint()
```

```
{
	CPaintDC dc(this); // device context for painting

	// TODO: Add your message handler code here

	// Do not call CWnd::OnPaint() for painting messages
}
```

COMMSCTRLDLG.H

```
#pragma once
#include "waverunner.h"
#include "CommsTab1.h"
#include "CommsTab2.h"
#include "CommsTab3.h"

// CCommsCtrlDlg

class CCommsCtrlDlg : public CPropertySheet
{
	DECLARE_DYNAMIC(CCommsCtrlDlg)

public:
	CCommsTab1 commsTab1;
	CCommsTab2 commsTab2;
	CCommsTab3 commsTab3;


	CCommsCtrlDlg(UINT nIDCaption, CWnd* pParentWnd = NULL, UINT
iSelectPage = 0);
	CCommsCtrlDlg(LPCTSTR pszCaption, CWnd* pParentWnd = NULL, UINT
iSelectPage = 0);
	virtual ~CCommsCtrlDlg();
	afx_msg void OnClose();

protected:
	DECLARE_MESSAGE_MAP()
};
```

COMMSCTRLDLG.CPP

```
// CommsCtrlDlg.cpp : implementation file
//

#include "stdafx.h"
#include "CommsCtrlDlg.h"


// CCommsCtrlDlg

IMPLEMENT_DYNAMIC(CCommsCtrlDlg, CPropertySheet)
CCommsCtrlDlg::CCommsCtrlDlg(UINT nIDCaption, CWnd* pParentWnd, UINT
iSelectPage)
	:CPropertySheet(nIDCaption, pParentWnd, iSelectPage)
{
```

```
}

CCommsCtrlDlg::CCommsCtrlDlg(LPCTSTR pszCaption, CWnd* pParentWnd, UINT
iSelectPage)
	:CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
	AddPage(&commsTab1);
	AddPage(&commsTab2);
	AddPage(&commsTab3);
	for (int channel=0; channel<8; channel++)
	{
		rxChannelInfo[channel].frequency=0;
		rxChannelInfo[channel].k=0;
		rxChannelInfo[channel].datarate=0;
		rxChannelInfo[channel].fileName="";
		txChannelInfo[channel].frequency=0;
		txChannelInfo[channel].k=0;
		txChannelInfo[channel].datarate=0;
		txChannelInfo[channel].fileName="";
	}
}

CCommsCtrlDlg::~CCommsCtrlDlg()
{
}

void CCommsCtrlDlg::OnClose()
{
	if (wr->rxTxEnable)
	{
		CString disp;
		disp="Cannot close panel while channels are active!";
		AfxMessageBox(disp,MB_OK,0);
	}
	else
	{
		CPropertySheet::OnClose();
	}
}

BEGIN_MESSAGE_MAP(CCommsCtrlDlg, CPropertySheet)
END_MESSAGE_MAP()


// CCommsCtrlDlg message handlers
```

COMMSTAB1.H

```
#pragma once
#include "Resource.h"


// CCommsTab1 dialog
class CCommsTab1 : public CPropertyPage
{
```

```
	DECLARE_DYNAMIC(CCommsTab1)

public:
	CCommsTab1();
	virtual ~CCommsTab1();

// Dialog Data
	enum { IDD = IDD_COMMSTAB1 };

protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support

	DECLARE_MESSAGE_MAP()
public:
	CEdit* configFile;
	CComboBox* txCombo;
	CComboBox* rxCombo;
	CButton* rxButton;
	CButton* txButton;
	CButton* rxTxButton;

	bool rxTxRunning;

	virtual BOOL OnInitDialog();
	afx_msg void OnCbnSelchangeRxchannelscombo();
	afx_msg void OnCbnSelchangeTxchannelscombo();
	afx_msg void OnBnClickedDfgfilefind();
	virtual BOOL OnSetActive();
	afx_msg void OnBnClickedRxenablebutton();
	afx_msg void OnBnClickedTxenablebutton();
	virtual void OnOK();
	afx_msg void OnBnClickedRxtxenablebutton();
	LRESULT OnThreadsFinished(WPARAM wparam, LPARAM lparam);
};
```

COMMSTAB1.CPP

```
// CommsTab1.cpp : implementation file
//

#include "stdafx.h"
#include "afxmt.h"
#include "direct.h"
#include "CommsCtrlDlg.h"
#include "CommsTab1.h"
#include "RxChannel.h"
#include "TxChannel.h"

// CCommsTab1 dialog

IMPLEMENT_DYNAMIC(CCommsTab1, CPropertyPage)
CCommsTab1::CCommsTab1()
	: CPropertyPage(CCommsTab1::IDD)
{
```

```
}

CCommsTab1::~CCommsTab1()
{
}

void CCommsTab1::DoDataExchange(CDataExchange* pDX)
{
	CPropertyPage::DoDataExchange(pDX);
}


BEGIN_MESSAGE_MAP(CCommsTab1, CPropertyPage)
	ON_CBN_SELCHANGE(IDC_RXCHANNELSCOMBO,
OnCbnSelchangeRxchannelscombo)
	ON_CBN_SELCHANGE(IDC_TXCHANNELSCOMBO,
OnCbnSelchangeTxchannelscombo)
	ON_BN_CLICKED(IDC_CFGFILEFIND, OnBnClickedDfgfilefind)
	ON_BN_CLICKED(IDC_RXTXENABLEBUTTON, OnBnClickedRxtxenablebutton)
	ON_MESSAGE(WM_PROCESSES_FINISHED,   OnThreadsFinished)
END_MESSAGE_MAP()


// CCommsTab1 message handlers

BOOL CCommsTab1::OnInitDialog()
{
	CPropertyPage::OnInitDialog();
	rxTxRunning=false;
	configFile = (CEdit*) GetDlgItem(IDC_CFGFILE);
	configFile->SetWindowText("D:\\WaveRadio\\config.wcf");

	sprintf(wr->configFile, "config.wcf");
	sprintf(wr->configPath, "D:\\WaveRadio");

	rxCombo = (CComboBox*) GetDlgItem(IDC_RXCHANNELSCOMBO);
	txCombo = (CComboBox*) GetDlgItem(IDC_TXCHANNELSCOMBO);
	rxTxButton = (CButton*) GetDlgItem(IDC_RXTXENABLEBUTTON);
	CString str;
	for (int i=0; i<9; i++)
	{
		str.Format(_T("%d"),i);
		rxCombo->AddString(str);
		txCombo->AddString(str);
	}
	txChannelStatus[0].status=(CEdit*) GetDlgItem(IDC_TX1STATUS);
	txChannelStatus[0].progress=(CProgressCtrl*)
GetDlgItem(IDC_TX1PROGRESS);
	txChannelStatus[1].status=(CEdit*) GetDlgItem(IDC_TX2STATUS);
	txChannelStatus[1].progress=(CProgressCtrl*)
GetDlgItem(IDC_TX2PROGRESS);
	txChannelStatus[2].status=(CEdit*) GetDlgItem(IDC_TX3STATUS);
	txChannelStatus[2].progress=(CProgressCtrl*)
GetDlgItem(IDC_TX3PROGRESS);
	txChannelStatus[3].status=(CEdit*) GetDlgItem(IDC_TX4STATUS);
	txChannelStatus[3].progress=(CProgressCtrl*)
GetDlgItem(IDC_TX4PROGRESS);
```

```
	txChannelStatus[4].status=(CEdit*) GetDlgItem(IDC_TX5STATUS);
	txChannelStatus[4].progress=(CProgressCtrl*)
GetDlgItem(IDC_TX5PROGRESS);
	txChannelStatus[5].status=(CEdit*) GetDlgItem(IDC_TX6STATUS);
	txChannelStatus[5].progress=(CProgressCtrl*)
GetDlgItem(IDC_TX6PROGRESS);
	txChannelStatus[6].status=(CEdit*) GetDlgItem(IDC_TX7STATUS);
	txChannelStatus[6].progress=(CProgressCtrl*)
GetDlgItem(IDC_TX7PROGRESS);
	txChannelStatus[7].status=(CEdit*) GetDlgItem(IDC_TX8STATUS);
	txChannelStatus[7].progress=(CProgressCtrl*)
GetDlgItem(IDC_TX8PROGRESS);
	rxChannelStatus[0].status=(CEdit*) GetDlgItem(IDC_RX1STATUS);
	rxChannelStatus[0].progress=(CProgressCtrl*)
GetDlgItem(IDC_RX1PROGRESS);
	rxChannelStatus[1].status=(CEdit*) GetDlgItem(IDC_RX2STATUS);
	rxChannelStatus[1].progress=(CProgressCtrl*)
GetDlgItem(IDC_RX2PROGRESS);
	rxChannelStatus[2].status=(CEdit*) GetDlgItem(IDC_RX3STATUS);
	rxChannelStatus[2].progress=(CProgressCtrl*)
GetDlgItem(IDC_RX3PROGRESS);
	rxChannelStatus[3].status=(CEdit*) GetDlgItem(IDC_RX4STATUS);
	rxChannelStatus[3].progress=(CProgressCtrl*)
GetDlgItem(IDC_RX4PROGRESS);
	rxChannelStatus[4].status=(CEdit*) GetDlgItem(IDC_RX5STATUS);
	rxChannelStatus[4].progress=(CProgressCtrl*)
GetDlgItem(IDC_RX5PROGRESS);
	rxChannelStatus[5].status=(CEdit*) GetDlgItem(IDC_RX6STATUS);
	rxChannelStatus[5].progress=(CProgressCtrl*)
GetDlgItem(IDC_RX6PROGRESS);
	rxChannelStatus[6].status=(CEdit*) GetDlgItem(IDC_RX7STATUS);
	rxChannelStatus[6].progress=(CProgressCtrl*)
GetDlgItem(IDC_RX7PROGRESS);
	rxChannelStatus[7].status=(CEdit*) GetDlgItem(IDC_RX8STATUS);
	rxChannelStatus[7].progress=(CProgressCtrl*)
GetDlgItem(IDC_RX8PROGRESS);
	for (int channel=0; channel<8;channel++)
	{
		txChannelStatus[channel].status->SetWindowText("Inactive");
		txChannelStatus[channel].progress->SetRange(0,100);
		txChannelStatus[channel].progress->SetPos(0);
		rxChannelStatus[channel].status->SetWindowText("Inactive");
		rxChannelStatus[channel].progress->SetRange(0,100);
		rxChannelStatus[channel].progress->SetPos(0);
	}
	rxCombo->SetCurSel(0);
	txCombo->SetCurSel(0);

	// TODO:  Add extra initialization here

	return TRUE;  // return TRUE unless you set the focus to a
control
	// EXCEPTION: OCX Property Pages should return FALSE
}

void CCommsTab1::OnCbnSelchangeRxchannelscombo()
{
```

```
	// TODO: Add your control notification handler code here
	wr->rxChannelsCount=rxCombo->GetCurSel();
	if (wr->rxChannelsCount>0)
	{
		if(wr->rxChannelsCount>wr->rxChannelsConfigured)
		{
			CString disp;
			disp.Format("Rx Channels %d - %d not configured.",
					wr->rxChannelsConfigured+1,wr-
>rxChannelsCount);
			AfxMessageBox(disp,MB_OK,0);
		}
	}
	if ((wr->rxChannelsCount+wr->txChannelsCount>0)
		& (wr->rxChannelsCount<=wr->rxChannelsConfigured)
		& (wr->txChannelsCount<=wr->txChannelsConfigured))
	{
		rxTxButton->EnableWindow(true);
	}
	else
	{
		rxTxButton->EnableWindow(false);
	}
}


void CCommsTab1::OnCbnSelchangeTxchannelscombo()
{
	// TODO: Add your control notification handler code here
	wr->txChannelsCount=txCombo->GetCurSel();
	if (wr->txChannelsCount>0)
	{
		if(wr->txChannelsCount>wr->txChannelsConfigured)
		{
			CString disp;
			disp.Format("Tx Channels %d - %d not configured.",
					wr->txChannelsConfigured+1,wr-
>txChannelsCount);
			AfxMessageBox(disp,MB_OK,0);
		}
	}
	if ((wr->rxChannelsCount+wr->txChannelsCount>0)
		& (wr->rxChannelsCount<=wr->rxChannelsConfigured)
		& (wr->txChannelsCount<=wr->txChannelsConfigured))
	{
		rxTxButton->EnableWindow(true);
	}
	else
	{
		rxTxButton->EnableWindow(false);
	}
}


void CCommsTab1::OnBnClickedDfgfilefind()
{
	// TODO: Add your control notification handler code here
```

```
	LPCSTR filefilter="WaveRunner Configuration files\0 *.WCF\0 All
files\0 *.*\0";
	CFileDialog filefind(true);
	filefind.m_ofn.lpstrTitle="Configuration file to retrieve";
	filefind.m_ofn.lpstrDefExt="WCF";
	filefind.m_ofn.lpstrFilter=filefilter;
	if (filefind.DoModal()==IDOK)
	{
		configFile->SetWindowText(filefind.GetPathName());
		CString buffer=filefind.GetFileName();
		for (int letter=0; letter<buffer.GetLength(); letter++)
		{
			wr->configFile[letter]=buffer[letter];
		}
		wr->configFile[letter]=0;
		int fileLength=buffer.GetLength();
		buffer=filefind.GetPathName();
		int pathLength=buffer.GetLength()-fileLength-1;
		int pos=0;
		for (int letter=0; letter<pathLength; letter++)
		{
			wr->configPath[pos]=buffer[letter];
			pos++;
			if (buffer[letter]==92)
			{
				wr->configPath[pos]=92;
				pos++;
			}
		}
		wr->configPath[pos]=0;
	}
}

BOOL CCommsTab1::OnSetActive()
{
	// TODO: Add your specialized code here and/or call the base
class
	int channelsCount, channelsConfigured;
	channelsCount=wr->txChannelsCount+wr->rxChannelsCount;
	channelsConfigured=wr->txChannelsConfigured+wr-
>rxChannelsConfigured;
	if (channelsCount>0)
	{
		if (channelsCount>channelsConfigured)
		{
			CString disp;
			disp.Format("Some Channels are not configured.");
			AfxMessageBox(disp,MB_OK,0);
			rxTxButton->EnableWindow(false);
		}
		else
		{
			rxTxButton->EnableWindow(true);
		}
	}
	else
	{
```

```
			rxTxButton->EnableWindow(false);
		}
		return CPropertyPage::OnSetActive();
}


void CCommsTab1::OnBnClickedRxtxenablebutton()
{
		// TODO: Add your control notification handler code here
		if (!rxTxRunning)
		{
			rxTxButton->EnableWindow(false);
			txCombo->EnableWindow(false);
			rxCombo->EnableWindow(false);
			wr->rxTxEnable=true;
			rxTxRunning=true;
			AfxBeginThread(mainRxTxThread,this);
			rxTxButton->SetWindowText("Stop Rx/Tx");
			rxTxButton->EnableWindow(true);
		}
		else
		{
			rxTxButton->EnableWindow(false);
			wr->rxTxEnable=false;
			for (int channel=0; channel<wr->rxChannelsCount; channel++)
			{
				rxBufferFull[channel].SetEvent();
			}
			for (int channel=0; channel<wr->txChannelsCount; channel++)
			{
				txBufferEmpty[channel].SetEvent();
			}
			MSG message;
			unsigned short threadsRunning=0;
			while(threadsRunning>0)
			{
				if (::PeekMessage(&message,NULL,0,0,PM_REMOVE))
				{
					::TranslateMessage(&message);
					::DispatchMessage(&message);
				}
				threadsRunning=wr->rxThreadsRunning+wr-
>txThreadsRunning;
			}
			txCombo->EnableWindow(true);
			rxCombo->EnableWindow(true);
			rxTxButton->SetWindowText("Start Rx/Tx");
			rxTxButton->EnableWindow(true);
			rxTxRunning=false;
		}
}


void CCommsTab1::OnOK()
{
		// TODO: Add your specialized code here and/or call the base
class
```

```
	if (rxTxRunning)
	{
			CString disp;
			disp.Format("Cannot close panel while channels are
active !!!");
			AfxMessageBox(disp,MB_OK,0);
	}
	else
	{
	  CPropertyPage::OnOK();
	}
}


LRESULT CCommsTab1::OnThreadsFinished(WPARAM wparam, LPARAM lparam)
{
	OnBnClickedRxtxenablebutton();
	return 0;
}
```

COMMSTAB2.H

```
#pragma once
#include "Resource.h"

// CCommsTab2 dialog

class CCommsTab2 : public CPropertyPage
{
	DECLARE_DYNAMIC(CCommsTab2)

	CEdit* rxFrequency[8];
	CComboBox* rxModulation [8];
	CEdit* rxSymbolRate[8];
	CEdit* rxFile[8];
	CButton* findRxFile[8];

	void SetRxChannelFileName(short rxChanNum);
	void GetControlPointers();
	void InitializeControls();
	void EnableControls();

public:
	CCommsTab2();
	virtual ~CCommsTab2();

// Dialog Data
	enum { IDD = IDD_COMMSTAB2 };

protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support

	DECLARE_MESSAGE_MAP()
public:
```

```
	virtual BOOL OnInitDialog();
	virtual BOOL OnSetActive();
	afx_msg void OnBnClickedFindfile1();
	afx_msg void OnBnClickedFindfile2();
	afx_msg void OnBnClickedFindfile3();
	afx_msg void OnBnClickedFindfile4();
	afx_msg void OnBnClickedFindfile5();
	afx_msg void OnBnClickedFindfile6();
	afx_msg void OnBnClickedFindfile7();
	afx_msg void OnBnClickedFindfile8();
	virtual BOOL OnKillActive();
};
```

COMMSTAB2.CPP

```
// CommsTab2.cpp : implementation file
//

#include "stdafx.h"
#include "CommsTab2.h"


// CCommsTab2 dialog

IMPLEMENT_DYNAMIC(CCommsTab2, CPropertyPage)
CCommsTab2::CCommsTab2()
	: CPropertyPage(CCommsTab2::IDD)
{
}

CCommsTab2::~CCommsTab2()
{
}

void CCommsTab2::DoDataExchange(CDataExchange* pDX)
{
	CPropertyPage::DoDataExchange(pDX);
}


BEGIN_MESSAGE_MAP(CCommsTab2, CPropertyPage)
	ON_BN_CLICKED(IDC_FINDFILE1, OnBnClickedFindfile1)
	ON_BN_CLICKED(IDC_FINDFILE2, OnBnClickedFindfile2)
	ON_BN_CLICKED(IDC_FINDFILE3, OnBnClickedFindfile3)
	ON_BN_CLICKED(IDC_FINDFILE4, OnBnClickedFindfile4)
	ON_BN_CLICKED(IDC_FINDFILE5, OnBnClickedFindfile5)
	ON_BN_CLICKED(IDC_FINDFILE6, OnBnClickedFindfile6)
	ON_BN_CLICKED(IDC_FINDFILE7, OnBnClickedFindfile7)
	ON_BN_CLICKED(IDC_FINDFILE8, OnBnClickedFindfile8)
END_MESSAGE_MAP()


// CCommsTab2 message handlers

BOOL CCommsTab2::OnInitDialog()
```

```
{
	CPropertyPage::OnInitDialog();

	// TODO:  Add extra initialization here
	GetControlPointers();
	InitializeControls();
	EnableControls();
	return TRUE;  // return TRUE unless you set the focus to a
control
	// EXCEPTION: OCX Property Pages should return FALSE
}


BOOL CCommsTab2::OnSetActive()
{
	// TODO: Add your specialized code here and/or call the base
class
	EnableControls();

	return CPropertyPage::OnSetActive();
}


void CCommsTab2::GetControlPointers()
{
	rxFrequency[0] = (CEdit*) GetDlgItem(IDC_FREQEDIT1);
	rxFrequency[1] = (CEdit*) GetDlgItem(IDC_FREQEDIT2);
	rxFrequency[2] = (CEdit*) GetDlgItem(IDC_FREQEDIT3);
	rxFrequency[3] = (CEdit*) GetDlgItem(IDC_FREQEDIT4);
	rxFrequency[4] = (CEdit*) GetDlgItem(IDC_FREQEDIT5);
	rxFrequency[5] = (CEdit*) GetDlgItem(IDC_FREQEDIT6);
	rxFrequency[6] = (CEdit*) GetDlgItem(IDC_FREQEDIT7);
	rxFrequency[7] = (CEdit*) GetDlgItem(IDC_FREQEDIT8);

	rxModulation[0] = (CComboBox*) GetDlgItem(IDC_MODULATION1);
	rxModulation[1] = (CComboBox*) GetDlgItem(IDC_MODULATION2);
	rxModulation[2] = (CComboBox*) GetDlgItem(IDC_MODULATION3);
	rxModulation[3] = (CComboBox*) GetDlgItem(IDC_MODULATION4);
	rxModulation[4] = (CComboBox*) GetDlgItem(IDC_MODULATION5);
	rxModulation[5] = (CComboBox*) GetDlgItem(IDC_MODULATION6);
	rxModulation[6] = (CComboBox*) GetDlgItem(IDC_MODULATION7);
	rxModulation[7] = (CComboBox*) GetDlgItem(IDC_MODULATION8);

	rxSymbolRate[0] = (CEdit*) GetDlgItem(IDC_SYMBOLRATE1);
	rxSymbolRate[1] = (CEdit*) GetDlgItem(IDC_SYMBOLRATE2);
	rxSymbolRate[2] = (CEdit*) GetDlgItem(IDC_SYMBOLRATE3);
	rxSymbolRate[3] = (CEdit*) GetDlgItem(IDC_SYMBOLRATE4);
	rxSymbolRate[4] = (CEdit*) GetDlgItem(IDC_SYMBOLRATE5);
	rxSymbolRate[5] = (CEdit*) GetDlgItem(IDC_SYMBOLRATE6);
	rxSymbolRate[6] = (CEdit*) GetDlgItem(IDC_SYMBOLRATE7);
	rxSymbolRate[7] = (CEdit*) GetDlgItem(IDC_SYMBOLRATE8);

	rxFile[0] = (CEdit*) GetDlgItem(IDC_DESTFILE1);
	rxFile[1] = (CEdit*) GetDlgItem(IDC_DESTFILE2);
	rxFile[2] = (CEdit*) GetDlgItem(IDC_DESTFILE3);
	rxFile[3] = (CEdit*) GetDlgItem(IDC_DESTFILE4);
	rxFile[4] = (CEdit*) GetDlgItem(IDC_DESTFILE5);
```

```
	rxFile[5] = (CEdit*) GetDlgItem(IDC_DESTFILE6);
	rxFile[6] = (CEdit*) GetDlgItem(IDC_DESTFILE7);
	rxFile[7] = (CEdit*) GetDlgItem(IDC_DESTFILE8);

	findRxFile[0] = (CButton*) GetDlgItem(IDC_FINDFILE1);
	findRxFile[1] = (CButton*) GetDlgItem(IDC_FINDFILE2);
	findRxFile[2] = (CButton*) GetDlgItem(IDC_FINDFILE3);
	findRxFile[3] = (CButton*) GetDlgItem(IDC_FINDFILE4);
	findRxFile[4] = (CButton*) GetDlgItem(IDC_FINDFILE5);
	findRxFile[5] = (CButton*) GetDlgItem(IDC_FINDFILE6);
	findRxFile[6] = (CButton*) GetDlgItem(IDC_FINDFILE7);
	findRxFile[7] = (CButton*) GetDlgItem(IDC_FINDFILE8);

}


void CCommsTab2::InitializeControls()
{
	CString fileName;
	for (int i=0;i<8;i++)
	{
		fileName.Format("D:\\WaveRadio\\RxChannel #%1u.wdf",i+1);
		rxFile[i]->SetWindowText(fileName);
		rxFrequency[i]->SetWindowText("23500000");
		rxModulation[i]->AddString("Test Tone");
		rxModulation[i]->AddString("QPSK");
		rxModulation[i]->AddString("8-PSK");
		rxModulation[i]->AddString("16-PSK");
		rxModulation[i]->SetCurSel(1);
		rxSymbolRate[i]->SetWindowText("4");
	}
}


void CCommsTab2::EnableControls()
{
	for (int i=0; i<wr->rxChannelsCount; i++)
	{
		rxFrequency[i]->EnableWindow(true);
		rxModulation[i]->EnableWindow(true);
		rxSymbolRate[i]->EnableWindow(true);
		rxFile[i]->EnableWindow(true);
		findRxFile[i]->EnableWindow(true);
	}
	for (int i=wr->rxChannelsCount; i<8; i++)
	{
		rxFrequency[i]->EnableWindow(false);
		rxModulation[i]->EnableWindow(false);
		rxSymbolRate[i]->EnableWindow(false);
		rxFile[i]->EnableWindow(false);
		findRxFile[i]->EnableWindow(false);
	}
}


void CCommsTab2::SetRxChannelFileName(short rxChanNum)
{
```

```
	LPCSTR filefilter="WaveRunner Data files\0 *.WDC\0 All files\0
*.*\0";
	CFileDialog filefind(false);
	filefind.m_ofn.lpstrTitle="File to save the received data";
	filefind.m_ofn.lpstrDefExt="WDC";
	filefind.m_ofn.lpstrFilter=filefilter;
	if (filefind.DoModal()==IDOK)
	{
		rxFile[rxChanNum]->SetWindowText(filefind.GetPathName());
	}
	else
	{
		rxFile[rxChanNum]->SetWindowText("");
	}
}


void CCommsTab2::OnBnClickedFindfile1()
{
	// TODO: Add your control notification handler code here
	SetRxChannelFileName(0);
}

void CCommsTab2::OnBnClickedFindfile2()
{
	// TODO: Add your control notification handler code here
	SetRxChannelFileName(1);
}

void CCommsTab2::OnBnClickedFindfile3()
{
	// TODO: Add your control notification handler code here
	SetRxChannelFileName(2);
}

void CCommsTab2::OnBnClickedFindfile4()
{
	// TODO: Add your control notification handler code here
	SetRxChannelFileName(3);
}

void CCommsTab2::OnBnClickedFindfile5()
{
	// TODO: Add your control notification handler code here
	SetRxChannelFileName(4);
}

void CCommsTab2::OnBnClickedFindfile6()
{
	// TODO: Add your control notification handler code here
	SetRxChannelFileName(5);
}

void CCommsTab2::OnBnClickedFindfile7()
{
	// TODO: Add your control notification handler code here
	SetRxChannelFileName(6);
```

```
}

void CCommsTab2::OnBnClickedFindfile8()
{
	// TODO: Add your control notification handler code here
	SetRxChannelFileName(7);
}

BOOL CCommsTab2::OnKillActive()
{
	// TODO: Add your specialized code here and/or call the base
class
	for (int rxCh=0; rxCh<wr->rxChannelsCount; rxCh++)
	{
		LPTSTR buffer=new char[80];
		rxFrequency[rxCh]->GetWindowText(buffer, 9);
		rxChannelInfo[rxCh].frequency=atol(buffer);
		rxChannelInfo[rxCh].k=rxModulation[rxCh]->GetCurSel()+1;
		rxSymbolRate[rxCh]->GetWindowText(buffer, 9);
		rxChannelInfo[rxCh].datarate=atol(buffer);
		rxFile[rxCh]->GetWindowText(buffer,80);
		rxChannelInfo[rxCh].fileName=buffer;
		if (wr->rxChannelsConfigured<wr->rxChannelsCount)
		{
		wr->rxChannelsConfigured=wr->rxChannelsCount;
		}
	}
	return CPropertyPage::OnKillActive();
}
```

COMMSTAB3.H

```
#pragma once
#include "Resource.h"

// CCommsTab3 dialog

class CCommsTab3 : public CPropertyPage
{
	DECLARE_DYNAMIC(CCommsTab3)

	CEdit* txFrequency[8];
	CComboBox* txModulation[8];
	CEdit* txSymbolRate[8];
	CEdit* txFile[8];
	CButton* findTxFile[8];
	CComboBox* attenuation[8];

	void SetTxChannelFileName(short txChanNum);
	void GetControlPointers();
	void InitializeControls();
	void EnableControls();

public:
	CCommsTab3();
```

```
	virtual ~CCommsTab3();

// Dialog Data
	enum { IDD = IDD_COMMSTAB3 };

protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support

	DECLARE_MESSAGE_MAP()
public:
	virtual BOOL OnInitDialog();
	virtual BOOL OnSetActive();
	afx_msg void OnBnClickedFilefind1();
	afx_msg void OnBnClickedFilefind2();
	afx_msg void OnBnClickedFilefind3();
	afx_msg void OnBnClickedFilefind4();
	afx_msg void OnBnClickedFilefind5();
	afx_msg void OnBnClickedFilefind6();
	afx_msg void OnBnClickedFilefind7();
	afx_msg void OnBnClickedFilefind8();
	virtual BOOL OnKillActive();
};
```

COMMSTAB3.CPP

```
// CommsTab3.cpp : implementation file
//

#include "stdafx.h"
#include "CommsTab3.h"
#include "math.h"


// CCommsTab3 dialog

IMPLEMENT_DYNAMIC(CCommsTab3, CPropertyPage)
CCommsTab3::CCommsTab3()
	: CPropertyPage(CCommsTab3::IDD)
{
}

CCommsTab3::~CCommsTab3()
{
}

void CCommsTab3::DoDataExchange(CDataExchange* pDX)
{
	CPropertyPage::DoDataExchange(pDX);
}


BEGIN_MESSAGE_MAP(CCommsTab3, CPropertyPage)
	ON_BN_CLICKED(IDC_FILEFIND1, OnBnClickedFilefind1)
	ON_BN_CLICKED(IDC_FILEFIND2, OnBnClickedFilefind2)
```

```
	ON_BN_CLICKED(IDC_FILEFIND3, OnBnClickedFilefind3)
	ON_BN_CLICKED(IDC_FILEFIND4, OnBnClickedFilefind4)
	ON_BN_CLICKED(IDC_FILEFIND5, OnBnClickedFilefind5)
	ON_BN_CLICKED(IDC_FILEFIND6, OnBnClickedFilefind6)
	ON_BN_CLICKED(IDC_FILEFIND7, OnBnClickedFilefind7)
	ON_BN_CLICKED(IDC_FILEFIND8, OnBnClickedFilefind8)
END_MESSAGE_MAP()


void CCommsTab3::GetControlPointers()
{
	txFrequency[0] = (CEdit*) GetDlgItem(IDC_TXFREQUENCY1);
	txFrequency[1] = (CEdit*) GetDlgItem(IDC_TXFREQUENCY2);
	txFrequency[2] = (CEdit*) GetDlgItem(IDC_TXFREQUENCY3);
	txFrequency[3] = (CEdit*) GetDlgItem(IDC_TXFREQUENCY4);
	txFrequency[4] = (CEdit*) GetDlgItem(IDC_TXFREQUENCY5);
	txFrequency[5] = (CEdit*) GetDlgItem(IDC_TXFREQUENCY6);
	txFrequency[6] = (CEdit*) GetDlgItem(IDC_TXFREQUENCY7);
	txFrequency[7] = (CEdit*) GetDlgItem(IDC_TXFREQUENCY8);

	txModulation[0] = (CComboBox*) GetDlgItem(IDC_TXMODULATION1);
	txModulation[1] = (CComboBox*) GetDlgItem(IDC_TXMODULATION2);
	txModulation[2] = (CComboBox*) GetDlgItem(IDC_TXMODULATION3);
	txModulation[3] = (CComboBox*) GetDlgItem(IDC_TXMODULATION4);
	txModulation[4] = (CComboBox*) GetDlgItem(IDC_TXMODULATION5);
	txModulation[5] = (CComboBox*) GetDlgItem(IDC_TXMODULATION6);
	txModulation[6] = (CComboBox*) GetDlgItem(IDC_TXMODULATION7);
	txModulation[7] = (CComboBox*) GetDlgItem(IDC_TXMODULATION8);

	txSymbolRate[0] = (CEdit*) GetDlgItem(IDC_TXDATARATE1);
	txSymbolRate[1] = (CEdit*) GetDlgItem(IDC_TXDATARATE2);
	txSymbolRate[2] = (CEdit*) GetDlgItem(IDC_TXDATARATE3);
	txSymbolRate[3] = (CEdit*) GetDlgItem(IDC_TXDATARATE4);
	txSymbolRate[4] = (CEdit*) GetDlgItem(IDC_TXDATARATE5);
	txSymbolRate[5] = (CEdit*) GetDlgItem(IDC_TXDATARATE6);
	txSymbolRate[6] = (CEdit*) GetDlgItem(IDC_TXDATARATE7);
	txSymbolRate[7] = (CEdit*) GetDlgItem(IDC_TXDATARATE8);

	txFile[0] = (CEdit*) GetDlgItem(IDC_TXFILE1);
	txFile[1] = (CEdit*) GetDlgItem(IDC_TXFILE2);
	txFile[2] = (CEdit*) GetDlgItem(IDC_TXFILE3);
	txFile[3] = (CEdit*) GetDlgItem(IDC_TXFILE4);
	txFile[4] = (CEdit*) GetDlgItem(IDC_TXFILE5);
	txFile[5] = (CEdit*) GetDlgItem(IDC_TXFILE6);
	txFile[6] = (CEdit*) GetDlgItem(IDC_TXFILE7);
	txFile[7] = (CEdit*) GetDlgItem(IDC_TXFILE8);

	findTxFile[0] = (CButton*) GetDlgItem(IDC_FILEFIND1);
	findTxFile[1] = (CButton*) GetDlgItem(IDC_FILEFIND2);
	findTxFile[2] = (CButton*) GetDlgItem(IDC_FILEFIND3);
	findTxFile[3] = (CButton*) GetDlgItem(IDC_FILEFIND4);
	findTxFile[4] = (CButton*) GetDlgItem(IDC_FILEFIND5);
	findTxFile[5] = (CButton*) GetDlgItem(IDC_FILEFIND6);
	findTxFile[6] = (CButton*) GetDlgItem(IDC_FILEFIND7);
	findTxFile[7] = (CButton*) GetDlgItem(IDC_FILEFIND8);

	attenuation[0] = (CComboBox*) GetDlgItem(IDC_ATTN1);
```

```
	attenuation[1] = (CComboBox*) GetDlgItem(IDC_ATTN2);
	attenuation[2] = (CComboBox*) GetDlgItem(IDC_ATTN3);
	attenuation[3] = (CComboBox*) GetDlgItem(IDC_ATTN4);
	attenuation[4] = (CComboBox*) GetDlgItem(IDC_ATTN5);
	attenuation[5] = (CComboBox*) GetDlgItem(IDC_ATTN6);
	attenuation[6] = (CComboBox*) GetDlgItem(IDC_ATTN7);
	attenuation[7] = (CComboBox*) GetDlgItem(IDC_ATTN8);
}


void CCommsTab3::InitializeControls()
{
	CString fileName;
	for (int i=0;i<8;i++)
	{
		fileName.Format("TxChannel #%1i.wdf",i+1);
		txFile[i]->SetWindowText(fileName);
		txFrequency[i]->SetWindowText("23500000");
		txModulation[i]->AddString("Test Tone");
		txModulation[i]->AddString("QPSK");
		txModulation[i]->AddString("8-PSK");
		txModulation[i]->AddString("16-PSK");
		txModulation[i]->SetCurSel(1);
		txSymbolRate[i]->SetWindowText("58125");
		char attn[3];
		for (int j=0;j<93;j=j+6)
		{
			sprintf(attn,"%2u",j);
			attenuation[i]->AddString(attn);
		}
		attenuation[i]->SetCurSel(2);
	}
}


void CCommsTab3::EnableControls()
{
	for (int i=0; i<wr->txChannelsCount; i++)
	{
		txFrequency[i]->EnableWindow(true);
		txModulation[i]->EnableWindow(true);
		txSymbolRate[i]->EnableWindow(true);
		txFile[i]->EnableWindow(true);
		findTxFile[i]->EnableWindow(true);
		attenuation[i]->EnableWindow(true);
	}
	for (int i=wr->txChannelsCount; i<8; i++)
	{
		txFrequency[i]->EnableWindow(false);
		txModulation[i]->EnableWindow(false);
		txSymbolRate[i]->EnableWindow(false);
		txFile[i]->EnableWindow(false);
		findTxFile[i]->EnableWindow(false);
		attenuation[i]->EnableWindow(false);
	}
}
```

```
void CCommsTab3::SetTxChannelFileName(short txChanNum)
{
	LPCSTR filefilter="WaveRunner Data files\0 *.WDF\0";
	CFileDialog filefind(true);
	filefind.m_ofn.lpstrTitle="File to retrieve the transmission
data";
	filefind.m_ofn.lpstrDefExt="WDF";
	filefind.m_ofn.lpstrFilter=filefilter;
	if (filefind.DoModal()==IDOK)
	{
		txFile[txChanNum]->SetWindowText(filefind.GetPathName());
	}
	else
	{
		txFile[txChanNum]->SetWindowText("");
	}
}

// CCommsTab3 message handlers

BOOL CCommsTab3::OnInitDialog()
{
	CPropertyPage::OnInitDialog();

	// TODO:  Add extra initialization here
	GetControlPointers();
	InitializeControls();
	EnableControls();

	return TRUE;  // return TRUE unless you set the focus to a
control
	// EXCEPTION: OCX Property Pages should return FALSE
}

BOOL CCommsTab3::OnSetActive()
{
	// TODO: Add your specialized code here and/or call the base
class
	EnableControls();

	return CPropertyPage::OnSetActive();
}


void CCommsTab3::OnBnClickedFilefind1()
{
	// TODO: Add your control notification handler code here
	SetTxChannelFileName(0);
}

void CCommsTab3::OnBnClickedFilefind2()
{
	// TODO: Add your control notification handler code here
	SetTxChannelFileName(1);
}

void CCommsTab3::OnBnClickedFilefind3()
```

```
{
	// TODO: Add your control notification handler code here
	SetTxChannelFileName(2);
}

void CCommsTab3::OnBnClickedFilefind4()
{
	// TODO: Add your control notification handler code here
	SetTxChannelFileName(3);

}

void CCommsTab3::OnBnClickedFilefind5()
{
	// TODO: Add your control notification handler code here
	SetTxChannelFileName(4);

}

void CCommsTab3::OnBnClickedFilefind6()
{
	// TODO: Add your control notification handler code here
	SetTxChannelFileName(5);

}

void CCommsTab3::OnBnClickedFilefind7()
{
	// TODO: Add your control notification handler code here
	SetTxChannelFileName(6);

}

void CCommsTab3::OnBnClickedFilefind8()
{
	// TODO: Add your control notification handler code here
	SetTxChannelFileName(7);

}

BOOL CCommsTab3::OnKillActive()
{
	// TODO: Add your specialized code here and/or call the base
class
	for (int txCh=0; txCh<wr->txChannelsCount; txCh++)
	{
		char* buffer=new char[80];
		txFrequency[txCh]->GetWindowText(buffer, 9);
		txChannelInfo[txCh].frequency=atol(buffer);
		txChannelInfo[txCh].k=txModulation[txCh]->GetCurSel()+1;
		txSymbolRate[txCh]->GetWindowText(buffer, 9);
		txChannelInfo[txCh].datarate=atol(buffer);
		txFile[txCh]->GetWindowText(buffer,80);
		txChannelInfo[txCh].fileName=buffer;
		attenuation[txCh]->GetWindowText(buffer,9);
		txChannelInfo[txCh].attenuation=pow(2, atol(buffer)/6);
		if (wr->txChannelsConfigured<wr->txChannelsCount)
```

```
            {
            wr->txChannelsConfigured=wr->txChannelsCount;
            }
      }
      return CPropertyPage::OnKillActive();
}
```

GLOBALVARS.H

```
#include "afxmt.h"
#include "WaveRunner.h"
      #define USERISR0
      //#define DEVICE_NUM 0

      #include "pmcradioi.h"
      //#include "pmcradio_memmap.h"

#ifndef TYPES_DEFINED
    #define TYPES_DEFINED

    #define WM_PROCESSES_FINISHED WM_USER + 1

    const float pi=3.1415926535;

      struct ChannelStatus
      {
            CEdit* status;
            CProgressCtrl* progress;
      };

      struct ChannelInfo{
            unsigned long frequency;
            unsigned short k;
            unsigned int datarate;
            CString fileName;
            unsigned short attenuation;
            bool FIFOInterruptMask;
            unsigned short FIFOInterruptStatus;
      };
#endif

//The one and only object of the WaveRunner card
extern WaveRunner* wr;

//Global Arrays storing the channel data and status
extern ChannelStatus txChannelStatus[8];
extern ChannelStatus rxChannelStatus[8];
extern ChannelInfo txChannelInfo[8];
extern ChannelInfo rxChannelInfo[8];

//The main and the Rx/Tx threads
extern UINT mainRxTxThread(LPVOID pParam);
extern UINT rxThread(LPVOID pParam);
extern UINT txThread(LPVOID pParam);
extern CString Modulate(int txChannelNumber);
```

```
extern void Demodulate(int rxChannelNumber, bool createLog);

extern CEvent txBufferEmpty[WaveRunner::maxChannels];
extern CEvent rxBufferFull[WaveRunner::maxChannels];
```

WAVERUNNER.H

```
#pragma once
#include "RxChannel.h"
#include "TxChannel.h"

//Static variable defining if the WaveRunner
//Singleton class has been initiated

class WaveRunner
{
private:
	WaveRunner(void);
	void InitVariables(void);

public:
	//The maximum number of channels
	const static unsigned short maxChannels=8;
	// The number of 32 bit samples per block
	const static unsigned short blockSize=1024;
	const static unsigned long rxClockFrequency=93000000;
	const static unsigned long txClockFrequency=93000000;

	unsigned long* firmwareRevisionDate;
	char configPath[80];
	char configFile[80];
	int cardStatus;
	bool configured;

	// DMA addresses
	unsigned long lDMAvAddress;
	unsigned long* DMA_virtual_Address;
	unsigned long lDMApAddress;
	unsigned long* DMA_physical_Address;
	unsigned long txControl;
	unsigned long rxControl;
	unsigned long interruptMask;
	unsigned long autoDMACtrl;
	unsigned long txFIFOmask;
	unsigned long rxFIFOmask;

	//Threads status
	unsigned int threadsReady;
	unsigned short rxThreadsRunning;
	unsigned short txThreadsRunning;

	// Status variables
	bool rxTxEnable;

	//Channel Pointers
```

```
        RxChannel* rxChannel[8];
        TxChannel* txChannel[8];

        //Channel Parameters
        unsigned long rxBlocksPerGroup;
        unsigned long rxGroupsPerChannel;
        unsigned long rxThresholdGroups;
        unsigned long rxChannelSize;
        unsigned long txBlocksPerGroup;
        unsigned long txGroupsPerChannel;
        unsigned long txThresholdGroups;
        unsigned long txChannelSize;
        unsigned long memorySize;

        unsigned long maxAmplitude;


        //Channels status
        unsigned short txChannelsCount;
        unsigned short rxChannelsCount;
        unsigned short rxChannelsConfigured;
        unsigned short txChannelsConfigured;

        static WaveRunner* getNewWaveRunner();
        int Open();
        int Close();
        int Configure();
        int enableTx(void);
        int disableTx(void);
        int enableRx(void);
        int disableRx(void);
        int enableRxTx();
        int disableRxTx();
        int enableInterrupts();
        int disableInterrupts();
        ~WaveRunner(void);
};
```

WAVERUNNER.CPP

```
#include "StdAfx.h"
#include "direct.h"
#include "math.h"
#include "WaveRunner.h"
#include "RxChannel.h"
#include "TxChannel.h"
#include "memory_map.h"


//Variable declaring if a WaveRunner object
//has already been created
bool bWaveRunnerAlreadyCreated=false;


//Class constructor
```

```
WaveRunner::WaveRunner(void)
{
	configFile[0]=0;
	configPath[0]=0;
	cardStatus=-1;
	configured=false;

	rxTxEnable=false;
	txChannelsCount=0;
	rxChannelsCount=0;
	txChannelsConfigured=0;
	rxChannelsConfigured=0;
	DMA_virtual_Address=&lDMAvAddress;
	DMA_physical_Address=&lDMApAddress;

	maxAmplitude=0x7FFF;
	InitVariables();
}


//Class destructor
WaveRunner::~WaveRunner(void)
{
}

void WaveRunner::InitVariables()
{
	threadsReady=0;
	rxThreadsRunning=0;
	txThreadsRunning=0;
}


//Singleton Class Instantiation
WaveRunner* WaveRunner::getNewWaveRunner()
{
	if (!bWaveRunnerAlreadyCreated)
	{
		bWaveRunnerAlreadyCreated=true;
		return new WaveRunner();
	}
	else
	{
		return NULL;
	}
}


int WaveRunner::Open()
{
	InitVariables();
	rxBlocksPerGroup=4;
	rxGroupsPerChannel=2;
	rxThresholdGroups=1;
	rxChannelSize=blockSize*rxBlocksPerGroup*rxGroupsPerChannel;
	txBlocksPerGroup=4;
	txGroupsPerChannel=2;
```

```
	txThresholdGroups=1;
	txChannelSize=blockSize*txBlocksPerGroup*txGroupsPerChannel;

	//Reset the DUC
	WriteWaveRunner(0x101FC, 0x2);
	WriteWaveRunner(0x103FC, 0x2);

	SetDMABufferSize(512);
	cardStatus=OpenMultiWaveRunner(0);
	if (!cardStatus)
	{
		//Disable Discrete output
		WriteWaveRunner(_DISCRETE_OUTPUT_CONTROL, 0x0);
		//Write 0 to DMA control register to make sure it's off
		WriteWaveRunner(_AUTO_DMA_CONTROL, 0x0);
		//Make sure all interrupts are disabled
		WriteWaveRunner(_GLOBAL_INTERRUPT_MASK,
_DISABLE_INTERRUPTS);
		WriteWaveRunner(_INTERRUPT_MASK, 0x0);
		//Get memory map pointers and set channels memory pointers
		GetDMAPA(&lDMApAddress, &lDMAvAddress);
		unsigned long totalMemory=GetMaxDMABufferSize();
		for (int channel=0; channel<maxChannels; channel++)
		{
			rxChannel[channel]=new RxChannel(channel);
			rxBufferFull[channel].ResetEvent();
			rxChannel[channel]->dataBuffer=
				(unsigned
long*)(lDMAvAddress+4*rxChannelSize*channel);
			for (int symbol=0; symbol<rxChannelSize; symbol++)
				*(rxChannel[channel]->dataBuffer+symbol)=0;
		}
		for (int channel=0; channel<maxChannels; channel++)
		{
			txChannel[channel]=new TxChannel(channel);
			txBufferEmpty[channel].ResetEvent();
			txChannel[channel]->dataBuffer=
				(unsigned
long*)(lDMAvAddress+4*rxChannelSize*maxChannels+

4*txChannelSize*channel);
			for (int symbol=0; symbol<rxChannelSize; symbol++)
				*(txChannel[channel]->dataBuffer+symbol)=0;
		}
		configured=false;
	}
	return cardStatus;
}


int WaveRunner::Close()
{
	if (!cardStatus)
	{
		//Make sure all interrupts are disabled
		WriteWaveRunner(_GLOBAL_INTERRUPT_MASK,
_DISABLE_INTERRUPTS);
```

```
            //Make sure that Tx and Rx are disabled
            disableRxTx();
            //Reset the DUC
            WriteWaveRunner(0x101FC, 0x1);
            for (int channel=0; channel<maxChannels; channel++)
            {
                  delete txChannel[channel];
            }
            for (int channel=0; channel<maxChannels; channel++)
            {
                  delete rxChannel[channel];
            }
            //Close card
            configured=false;
            cardStatus=-1;
            return CloseMultiWaveRunner(0);
      }
      else
      {
            return -2;
      }
}


//Configuration subroutine
int WaveRunner::Configure()
{
      unsigned long writeData=0x0;
      if ((rxChannelsCount==0) & (txChannelsCount==0))
      {
            AfxMessageBox("No Tx/Rx channels specified", MB_OK,0);
            return -4;
      }
      if (configFile==NULL)
      {
            AfxMessageBox("No configuration file specified", MB_OK,0);
            return -5;
      }
      //Configure the Up and Downconverters, using the files produced
      //by the Configuration Tool
      _chdir(configPath);
      int ioError=ConfigWaveRunner(configFile);
      if (ioError)
      {
            AfxMessageBox("Unable to configure WaveRunner", MB_OK,0);
            return ioError;
      }
      //Read Tx and Rx status
      ReadWaveRunner(_TRANSMIT_CONTROL, &txControl);
      txControl=txControl & 0xFFFFFF8F;
      ReadWaveRunner(_RECEIVE_CONTROL, &rxControl);
      rxControl=rxControl & 0xFFFFFF8F;
      ReadWaveRunner(_INTERRUPT_MASK, &interruptMask);
      interruptMask=interruptMask & 0xFFF00000;
      ReadWaveRunner(_AUTO_DMA_CONTROL, &autoDMACtrl);
      autoDMACtrl=autoDMACtrl & 0xfffffc0f;
      txFIFOmask=0;
```

```
	rxFIFOmask=0;
	//Make sure that Tx and Rx are disabled
	WriteWaveRunner(_TRANSMIT_CONTROL, 0x0);
	WriteWaveRunner(_RECEIVE_CONTROL,0x0);
	WriteWaveRunner(_INTERRUPT_MASK, 0x0);


	//********* Receiver Configuration ***************

	//Write the Rx DMA Control Register in the PCI configuration
	//address space
	char PCIConfig[4];
	PCIConfig[0]=0x7E;
	PCIConfig[1]=char((blockSize/2) & 0xFF);
	PCIConfig[2]=char(((blockSize/2) & 0xFF00)>>8);
	PCIConfig[3]=char(((blockSize/2) & 0xFF0000)>>16);
	WriteWRConfigSpace(0x4C, PCIConfig, 4);
	//For every Rx channel
	for (int rxCh=0; rxCh<maxChannels; rxCh++)
	{
		//Write the Auto DMA Address registers
		unsigned long rxAddress=lDMApAddress+4*rxChannelSize*rxCh;
		WriteWaveRunner(_RX_MEMORY_AREA_0_ADDRESS+0x10*rxCh,
rxAddress);
		//Write Auto DMA block count register
		WriteWaveRunner(_RX_MEMORY_AREA_0_BLOCK_COUNT+0x10*rxCh,
rxBlocksPerGroup);
		//Write the Auto DMA Group Count Registers
		WriteWaveRunner(_RX_MEMORY_AREA_0_GROUP_COUNT+0x10*rxCh,
rxGroupsPerChannel);
		//Write the memory area sizes

	writeData=(rxChannelSize*rxThresholdGroups/(2*rxGroupsPerChannel)
) |
				((rxChannelSize/2)<<16);
		WriteWaveRunner(_RX_MEMORY_AREA_0_LIMITS+0x10*rxCh,
writeData);
		//Write the memory area limits
		unsigned long startOffset=rxCh*rxChannelSize/2;
		unsigned long endOffset=(rxCh+1)*rxChannelSize/2-1;
		writeData=startOffset|(endOffset<<16);
		WriteWaveRunner(_RX_MEMORY_AREA_0_POINTER+0x10*rxCh,
writeData);
		//Write the memory organization control register
		unsigned short channelMask=(1<<rxCh);
		writeData=(rxCh<<12)|(rxCh<<8);
		if (rxCh<rxChannelsCount)
			writeData=writeData | channelMask;
		WriteWaveRunner(_RX_MEMORY_AREA_0_ORGANIZATION+0x10*rxCh,
writeData);
		//Mofdify the Interrupt Mask
		if (rxCh<rxChannelsCount)
		{
		interruptMask=interruptMask | (0x1 << (4+rxCh));
			rxFIFOmask=rxFIFOmask | (0xB << (4*rxCh));
		}
	}
```

```
      //Write Rx FIFO Interrupt Mask
      WriteWaveRunner(_RECEIVE_FIFO_INTERRUPT_MASK, rxFIFOmask);
      //Flush Rx FIFOs
      WriteWaveRunner(_RECEIVE_CONTROL, rxControl | _FIFO_FLUSH);
      WriteWaveRunner(_RECEIVE_CONTROL, rxControl & _TX_FIFO_ENABLE);
      //Configure Receiver to 8X1 channels
      writeData=rxControl | _BIT_REGISTERS_ENABLE |
                  _RX_MASTER_ENABLE | _RX_CIRCUITRY_ENABLE;
      if (rxChannelsCount>0) writeData=writeData | ((rxChannelsCount-
1)<<4);
      WriteWaveRunner(_RECEIVE_CONTROL, writeData);
      //Disable Timing Control
      WriteWaveRunner(_RECEIVE_TIMING_CONTROL, 0x0);
      //Set Receive clock frequency
      //WriteWaveRunner(0x001128, clockFrequency-2);

      //*************** Transmiter Configuration ********************

      //Write the Tx DMA Control Register in the PCI configuration
address space
      PCIConfig[0]=0x6E;
      PCIConfig[1]=char((blockSize/2) & 0xFF);
      PCIConfig[2]=char(((blockSize/2) & 0xFF00)>>8);
      PCIConfig[3]=char(((blockSize/2) & 0xFF0000)>>16);
      WriteWRConfigSpace(0x54, PCIConfig, 4);
      for (int txCh=0; txCh<maxChannels; txCh++)
      {
            //Write the Auto DMA Address registers

      writeData=lDMApAddress+4*(rxChannelSize*maxChannels+txChannelSize
*txCh);
            WriteWaveRunner(_TX_MEMORY_AREA_0_ADDRESS+0x10*txCh,
writeData);
            //Write Auto DMA block count register
            WriteWaveRunner(_TX_MEMORY_AREA_0_BLOCK_COUNT+0x10*txCh,
txBlocksPerGroup);
        //Write the Auto DMA Group Count Registers
            WriteWaveRunner(_TX_MEMORY_AREA_0_GROUP_COUNT+0x10*txCh,
txGroupsPerChannel);
            //Write the memory area sizes
            writeData=(blockSize*txBlocksPerGroup*txThresholdGroups/2)
                                    | ((txChannelSize/2)<<16);
            WriteWaveRunner(_TX_MEMORY_AREA_0_LIMITS+0x10*txCh,
writeData);
            //Write the memory area limits
            unsigned long startOffset=txCh*txChannelSize/2;
            unsigned long endOffset=(txCh+1)*txChannelSize/2-1;
            writeData=startOffset | (endOffset<<16);
            WriteWaveRunner(_TX_MEMORY_AREA_0_POINTER+0x10*txCh,
writeData);

            //Write the memory organization control register
            writeData=(txCh<<12)|(txCh<<8);
            if (txCh<txChannelsCount)
                  writeData=writeData | (1<<txCh);
            WriteWaveRunner(_TX_MEMORY_AREA_0_ORGANIZATION+0x10*txCh,
writeData);
```

```
            //Write the FIFO interrupt Mask
            WriteWaveRunner(_TRANSMIT_FIFO_INTERRUPT_MASK, 0);
            //Write the Interrupt Mask register
            if (txCh<txChannelsCount)
            {
            interruptMask=interruptMask | (0x1 << (12+txCh));
                  txFIFOmask=txFIFOmask | (0xB << (4*txCh));
            }
      }
      //Write Tx FIFO interrupt mask
      WriteWaveRunner(_TRANSMIT_FIFO_INTERRUPT_MASK, txFIFOmask);
      //Flush device FIFOs;
      ReadWaveRunner(_TRANSMIT_CONTROL, &writeData);
      WriteWaveRunner(_TRANSMIT_CONTROL, writeData | _TX_FIFO_FLUSH);
      WriteWaveRunner(_TRANSMIT_CONTROL, writeData & _TX_FIFO_ENABLE);
      //Configure Transmitter to 8X1 channels
      writeData=txControl | _TX_BIT_REGISTERS_ENABLE |
                  _TX_MASTER_ENABLE | _TX_MASTER_SYNC_ENABLE |
_TX_CIRCUITRY_ENABLE;
      if (txChannelsCount>0) writeData=writeData | ((txChannelsCount-
1)<<4);
      WriteWaveRunner(_TRANSMIT_CONTROL, writeData);
      //Disable Timing Control
      WriteWaveRunner(_TX_TIMING_CONTROL, 0x0);
      //Set Transmit Clock Rate
      //WriteWaveRunner(0x002128, clockFrequency-2);
      //Disable the PRN function
      //WriteWaveRunner(_PRN_CONTROL, 0x1A);
      //WriteWaveRunner(_PRN_ZERO_IQ_VALUE, 0x0);
      //WriteWaveRunner(_PRN_ONE_IQ_VALUE, 0x0);

      //Write interrupt mask
      WriteWaveRunner(_INTERRUPT_MASK, interruptMask);
      //Write Auto DMA control
      writeData=((blockSize/2)<<16);
      if (txChannelsCount>0) writeData=writeData | ((txChannelsCount-
1)<<7);
      if (rxChannelsCount>0) writeData=writeData | ((rxChannelsCount-
1)<<4);
      writeData=writeData | _AUTO_COUNTERS_RELOAD;
      WriteWaveRunner(_AUTO_DMA_CONTROL, writeData);
      //Perform dummy reads in order to clear status
      unsigned long Dummy;
      ReadWaveRunner(_RECEIVE_FIFO_INTERRUPT_STATUS, &Dummy);
      ReadWaveRunner(_TRANSMIT_FIFO_INTERRUPT_STATUS, &Dummy);
      ReadWaveRunner(_INTERRUPT_MASK, &Dummy);
      configured=true;
      return 0;
}


int WaveRunner::enableTx(void)
{
      if (configured)
      {
            unsigned long readData;
            //Set Auto DMA Control Register
```

```
            ReadWaveRunner(_AUTO_DMA_CONTROL,&readData);
            WriteWaveRunner(_AUTO_DMA_CONTROL, readData |
_TX_AUTO_DMA_ENABLE);
            //Set Interrupt Mask Register
            ReadWaveRunner(_INTERRUPT_MASK, &readData);
            readData=readData |  _DMA_ABORT_DETECTED_ENABLE |
                        _TX_DMA_COMPLETE_ENABLE |
_TX_FIFO_INTERRUPT_ENABLE;
            WriteWaveRunner(_INTERRUPT_MASK, readData);
            //Enable Interrupts
            WriteWaveRunner(_GLOBAL_INTERRUPT_MASK,
_ENABLE_INTERRUPTS);
            //Set Tx Control Register
            ReadWaveRunner(_TRANSMIT_CONTROL, &readData);
            WriteWaveRunner(_TRANSMIT_CONTROL, readData | _TX_ENABLE);
            //Send master sync to DUC
            for (int i=0; i<500; i++);
            WriteWaveRunner(0x2130,0x1);
            return 0;
      }
      else
      {
            return -4;
      }
}

int WaveRunner::disableTx(void)
{
      if (configured)
      {
            //Disable interrupts
            WriteWaveRunner(_GLOBAL_INTERRUPT_MASK,
_DISABLE_INTERRUPTS);
            //WriteWaveRunner(_TRANSMIT_FIFO_INTERRUPT_MASK,
_DISABLE_INTERRUPTS);
            unsigned long readData;
            ReadWaveRunner(_INTERRUPT_MASK, &readData);
            WriteWaveRunner(_INTERRUPT_MASK, readData &
_TX_INTERRUPTS_DISABLE);
            //Disable DMA
            ReadWaveRunner(_AUTO_DMA_CONTROL, &readData);
            WriteWaveRunner(_AUTO_DMA_CONTROL, readData &
_TX_AUTO_DMA_DISABLE);
            //Disable Receiver circuitry
            ReadWaveRunner(_TRANSMIT_CONTROL, &readData);
            WriteWaveRunner(_TRANSMIT_CONTROL, readData & _TX_DISABLE);
            //Reset the DUC
            WriteWaveRunner(0x101FC, 0x2);
            WriteWaveRunner(0x103FC, 0x2);
            return 0;
      }
      else
      {
            return -4;
      }
}
```

```
int WaveRunner::enableRx(void)
{
	if (configured)
	{
		unsigned long readData;
		//Set Auto DMA Control Register
		ReadWaveRunner(_AUTO_DMA_CONTROL,&readData);
		WriteWaveRunner(_AUTO_DMA_CONTROL, readData |
_RX_AUTO_DMA_ENABLE);
		//Set Interrupt Mask Register
		ReadWaveRunner(_INTERRUPT_MASK, &readData);
		WriteWaveRunner(_INTERRUPT_MASK, readData |
_RX_DMA_COMPLETE_ENABLE);
		//Enable Interrupts
		WriteWaveRunner(_GLOBAL_INTERRUPT_MASK,
_ENABLE_INTERRUPTS);
		//Set Receive Control register
		ReadWaveRunner(_RECEIVE_CONTROL, &readData);
		WriteWaveRunner(_RECEIVE_CONTROL, readData | _RX_ENABLE);
		return 0;
	}
	else
	{
		return -4;
	}
}


int WaveRunner::disableRx(void)
{
	if (configured)
	{
		//Disable interrupts
		WriteWaveRunner(_GLOBAL_INTERRUPT_MASK,
_DISABLE_INTERRUPTS);
		//WriteWaveRunner(_RECEIVE_FIFO_INTERRUPT_MASK,
_DISABLE_INTERRUPTS);
		unsigned long readData;
	    ReadWaveRunner(_INTERRUPT_MASK, &readData);
		WriteWaveRunner(_INTERRUPT_MASK, readData &
_RX_INTERRUPTS_DISABLE);
		//Disabe DMA
		ReadWaveRunner(_AUTO_DMA_CONTROL, &readData);
		WriteWaveRunner(_AUTO_DMA_CONTROL, readData &
_RX_AUTO_DMA_DISABLE);
		//Disable Receiver circuitr
		ReadWaveRunner(_RECEIVE_CONTROL, &readData);
		WriteWaveRunner(_RECEIVE_CONTROL, readData & _RX_DISABLE);
		return 0;
	}
	else
	{
		return -4;
	}
}
```

```
int WaveRunner::enableRxTx()
{
	if (configured)
	{
		if (rxChannelsCount>0) enableRx();
		for (int i=0; i<5000;i++);
		if (txChannelsCount>0) enableTx();
		return 0;
	}
	else
	{
		return -4;
	}
}


int WaveRunner::disableRxTx()
{
	if (configured)
	{
		disableInterrupts();
		if (rxChannelsCount>0) disableRx();
		if (txChannelsCount>0) disableTx();
		return 0;
	}
	else
	{
		return -4;
	}
}


int WaveRunner::enableInterrupts()
{
	if (configured)
	{
		WriteWaveRunner(_GLOBAL_INTERRUPT_MASK,
_ENABLE_INTERRUPTS);
		return 0;
	}
	else
	{
		return -4;
	}
}


int WaveRunner::disableInterrupts()
{
	if (configured)
	{
		WriteWaveRunner(_GLOBAL_INTERRUPT_MASK,
_DISABLE_INTERRUPTS);
		return 0;
	}
	else
```

```
	{
		return -4;
	}
}
```

WAVERUNNERCHANNEL.H

```
#pragma once

class WaveRunnerChannel
{
public:
	unsigned short channelNumber;
	unsigned short channelOffset;
	unsigned long frequency;
	unsigned short k;

	CString dataFileName;

	unsigned long dataRate;
	unsigned long offsetAddress;
	unsigned long* dataBuffer;
	unsigned int groupsTransfered;
	unsigned short groupCount;
	//bool bufferReady;

	bool terminateProcess;
	bool threadRunning;
	bool threadReady;

	WaveRunnerChannel();
	~WaveRunnerChannel(void);
};
```

WAVERUNNERCHANNEL.CPP

```
#include "StdAfx.h"
#include "waverunnerchannel.h"

WaveRunnerChannel::WaveRunnerChannel(){};

WaveRunnerChannel::~WaveRunnerChannel(void)
{
}
```

RXCHANNEL.H

```
#pragma once
#include "waverunnerchannel.h"

class RxChannel :
```

```
      public WaveRunnerChannel
{
public:
      unsigned int groupsSaved;

      RxChannel(unsigned short chanNum=0,
                  unsigned long freq=23500000,
                    unsigned short k=2,
                    unsigned long dataRate=50000,
                    CString dfile="");
      ~RxChannel(void);
      int setFrequency(unsigned long frequency);
};
```

RXCHANNEL.CPP

```
#include "StdAfx.h"
#include "math.h"
#include "rxchannel.h"
#include "Memory_Map.h"


RxChannel::RxChannel(unsigned short chanNum,
                              unsigned long freq,
                              unsigned short kmod,
                              unsigned long dRate,
                              CString dFile)
{
      channelNumber=chanNum;
      frequency=freq;
      k=kmod;
      dataRate=dRate;
      dataFileName=dFile;
      dataBuffer=NULL;
      groupsTransfered=0;
      threadRunning=false;
      terminateProcess=true;
      threadReady=false;
      groupCount=0;
      if (channelNumber<4)
      {
            offsetAddress=0x40000;
            channelOffset=channelNumber;
      }
      else
      {
            offsetAddress=0x80000;
            channelOffset=channelNumber-4;
      }
      setFrequency(freq);
      groupsSaved=0;
}


RxChannel::~RxChannel(void)
```

```
{
}


int RxChannel::setFrequency(unsigned long freq)
{
      frequency=freq;

      int ioError=-4;
      if (wr->configured)
      {
            unsigned long
freqAddress=offsetAddress+4*(0x1000*channelOffset+5);
            unsigned long freqValue=unsigned
long(freq*pow(2,32)/WaveRunner::rxClockFrequency);
            ioError=WriteWaveRunner(freqAddress, freqValue);
            if (!ioError)
            {
                  ioError=WriteWaveRunner(freqAddress+4, freqValue &
0x1);
            }
      }
      return ioError;
}
```

TXCHANNEL.H

```
#pragma once
#include "waverunnerchannel.h"

class TxChannel :
      public WaveRunnerChannel
{
public:
      unsigned int groupsLoaded;
      unsigned short attenuation;

      ~TxChannel(void);
      TxChannel(unsigned short chanNum=0,
                  unsigned long freq=23500000,
                    unsigned short k=2,
                    unsigned long dataRate=50000,
                    CString dfile="");
      int setFrequency(unsigned long frequency);
      int setDataRate(unsigned long dataRate);
};
```

TXCHANNEL.CPP

```
#include "StdAfx.h"
#include "math.h"
#include "GlobalVars.h"
#include "txchannel.h"
```

```
//#include "pmcradioi.h"


TxChannel::TxChannel(unsigned short chanNum,
                  unsigned long freq,
                    unsigned short kmod,
                    unsigned long dRate,
                    CString dFile)
{
      channelNumber=chanNum;
      k=kmod;
      dataFileName=dFile;
      dataBuffer=NULL;
      groupsTransfered=0;
      threadRunning=false;
      terminateProcess=true;
      threadReady=false;
      groupCount=0;
      attenuation=1;
      if (channelNumber<4)
      {
            offsetAddress=0x10000;
            channelOffset=channelNumber;
      }
      else
      {
            offsetAddress=0x10200;
            channelOffset=channelNumber-4;
      }
      setFrequency(freq);
      setDataRate(dRate);
      groupsLoaded=0;
}

TxChannel::~TxChannel(void)
{
}


int TxChannel::setFrequency(unsigned long freq)
{
      int ioError=-4;
      frequency=freq;
      if (wr->configured)
      {
            unsigned long
freqAddress=offsetAddress+4*(0x20*channelOffset+0x8);
            unsigned long freqValue=unsigned
long(freq*pow(2,32)/WaveRunner::txClockFrequency);
            unsigned long lfreq, ufreq;
            ufreq=(freqValue & 0xFFFF0000)>>16;
            lfreq=freqValue & 0xFFFF;
            ioError=WriteWaveRunner(freqAddress, ufreq);
            if (!ioError)
            {
                  ioError=WriteWaveRunner(freqAddress+4, lfreq);
            }
```

```
	}
	return ioError;
}


int TxChannel::setDataRate(unsigned long dRate)
{
	dataRate=dRate;
	int ioError=-4;
	if (wr->configured)
	{
		unsigned long
freqAddress=offsetAddress+4*(0x20*channelOffset+0x4);
		unsigned _int64 symbolRate=unsigned _int64(dataRate);
		unsigned _int64 mult=unsigned _int64(pow(2,48));
		double
divisor=double(pow(2,48))*dataRate/WaveRunner::txClockFrequency;
		unsigned _int64 dataValue=unsigned _int64(divisor);
		unsigned long lfreq, mfreq, ufreq;
		lfreq=dataValue & 0xFFFF;
		mfreq=(dataValue & 0xFFFF0000)>>16;
		ufreq=(dataValue & 0xFFFF00000000)>>32;
		ioError=WriteWaveRunner(freqAddress, ufreq);
		if (!ioError)
		{
			ioError=WriteWaveRunner(freqAddress+4, mfreq);
			if (!ioError)
			{
				ioError=WriteWaveRunner(freqAddress+8, lfreq);
			}
		}
	}
	return ioError;
}
```

WAVERUNNERISR.CPP

```
#include "StdAfx.h"
#include "afxmt.h"
#include "Math.h"
#include "direct.h"
#include "RxChannel.h"
#include "TxChannel.h"
#include "Memory_map.h"
#include "resource.h"

//Define and initialize Global Structures and variables
ChannelStatus txChannelStatus[8];
ChannelStatus rxChannelStatus[8];
ChannelInfo txChannelInfo[8];
ChannelInfo rxChannelInfo[8];

CCriticalSection cSection;
CEvent allChannelsReady, allChannelsDone;
CEvent txBufferEmpty[WaveRunner::maxChannels],
```

```
        rxBufferFull[WaveRunner::maxChannels];


//************************************************************
// Interrupt Service Routine. The contents of the Interrupt
// Status Register are stored in the variable "Status". Also,
// interrupts have been disabled, so we need to re-enable them
// before exiting the routine.
//************************************************************

void PMCRadioIsr0(unsigned long status)
{
      //Determine if interrupt is due to the card
      if ((status & _GLOBAL_INTERRUPT)!=0)
      {
            // If interrupt is due to a Rx FIFO interrupt
            if ((status & 0x1)!=0)
            {
                  unsigned long FIFOStatus;
                  unsigned short rxChFIFO;
                  //Read Rx FIFO Interrupt Status
            ReadWaveRunner(_RECEIVE_FIFO_INTERRUPT_STATUS,
&FIFOStatus);
                  //For each rx channel
                  for (int rxCh=0; rxCh<wr->rxChannelsCount; rxCh++)
                  {
                        //Check to see if the channel has caused the
interrupt
                        rxChFIFO=(FIFOStatus & (0xF << (4*rxCh))) >>
(4*rxCh);
                        if (rxChFIFO!=0)
                        {

      rxChannelInfo[rxCh].FIFOInterruptStatus=true;

      rxChannelInfo[rxCh].FIFOInterruptMask=rxChFIFO;
                        }
                  }
            }
            // If interrupt is due to a receive channel DMA complete
            if ((status & 0x4)!=0)
            {
                  //For every rxChannel
                  for (int rxCh=0; rxCh<wr->rxChannelsCount; rxCh++)
                  {
                        //If the channel was the cause of the interrupt
                        if((status & (0x1<< (4+rxCh)))!=0)
                        {
                              //increase the channel counter of
transfered blocks
                              wr->rxChannel[rxCh]->groupsTransfered++;
                              unsigned long buffer;
                              if (rxCh<4)
                              {
                                    ReadWaveRunner(0x10, &buffer);
                              }
                              else
```

```
                                {
                                      ReadWaveRunner(0x14, &buffer);
                                }
                                unsigned short ch=rxCh;
                                if (rxCh>3){ch=ch-4;}
                                wr->rxChannel[rxCh]->groupCount=((buffer
& (0xFF << (8*ch))) >> (8*ch));
                                //Set event to wake up channel thread
                                rxBufferFull[rxCh].SetEvent();
                          }
                    }
              }
              // If interrupt is due to a Tx FIFO interrupt
              if ((status & 0x2)!=0)
              {
                    unsigned long FIFOStatus;
                    unsigned short txChFIFO;
                    //Read Rx FIFO Interrupt Status
              ReadWaveRunner(_TRANSMIT_FIFO_INTERRUPT_STATUS,
&FIFOStatus);
                    //For each rx channel
                    for (int txCh=0; txCh<wr->txChannelsCount; txCh++)
                    {
                          //Check to see if the channel has caused the
interrupt
                          txChFIFO=(FIFOStatus & (0xF << (4*txCh))) >>
(4*txCh);
                          if (txChFIFO!=0)
                          {

      txChannelInfo[txCh].FIFOInterruptStatus=true;

      txChannelInfo[txCh].FIFOInterruptMask=txChFIFO;
                          }
                    }
              }
              // If interrupt is due to a transmit channel DMA complete
              if ((status & 0x8)!=0)
              {
                    //For every txChannel
                    for (int txCh=0; txCh<wr->txChannelsCount; txCh++)
                    {
                          //If the channel was the cause of the interrupt
                          if((status & (0x1<< (12+txCh)))!=0)
                          {
                                //increase the channel counter of
transfered blocks
                                wr->txChannel[txCh]->groupsTransfered++;
                                unsigned long buffer;
                                if (txCh<4)
                                {
                                      ReadWaveRunner(0x18, &buffer);
                                }
                                else
                                {
                                      ReadWaveRunner(0x1C, &buffer);
                                }
```

```
                              unsigned short ch=txCh;
                              if (txCh>3){ch=ch-4;}
                              wr->txChannel[txCh]->groupCount=((buffer
& (0xFF << (8*ch))) >> (8*ch));
                              txBufferEmpty[txCh].SetEvent();
                        }
                  }
            }
      }
      //Enable interrupts
      WriteWaveRunner(_GLOBAL_INTERRUPT_MASK, 0x1);
}

//***********************************************
// This is the main "parent" thread which controls
// the Rx and Tx channels threads.
//***********************************************

UINT mainRxTxThread(LPVOID pParam)
{
      CWnd* parentWindow = (CWnd*) pParam;

      //As a first step, allocate momory space for the channels
      wr->threadsReady=0;

      //Try to configure the card
    int error=wr->Configure();
      if (error)
      {
            wr->Close();
            CString disp;
            disp="WaveRunner not properly configured.\n Process will
abort.";
            return error;
      }

      // Configure channels by passing the parameters stored in the
..info tables
      // Then start channels threads

      for (int channel=0;channel<wr->maxChannels;channel++)
      {
            if (channel<wr->txChannelsCount)
            {
                  txChannelStatus[channel].status-
>SetWindowText("Initializing");
                  wr->txChannel[channel]-
>setFrequency(txChannelInfo[channel].frequency);
                  wr->txChannel[channel]->k=txChannelInfo[channel].k;
                  wr->txChannel[channel]-
>setDataRate(txChannelInfo[channel].datarate);
                  wr->txChannel[channel]-
>attenuation=txChannelInfo[channel].attenuation;
                  txChannelInfo[channel].FIFOInterruptStatus=false;
                  txChannelInfo[channel].FIFOInterruptMask=0;
                  char fName[80];
                  int pos=0;
```

```
                  for (int letter=0;
letter<txChannelInfo[channel].fileName.GetLength(); letter++)
                  {

      fName[pos]=txChannelInfo[channel].fileName[letter];
                        pos++;
                        if
(txChannelInfo[channel].fileName[letter]==92)
                        {
                              fName[pos]=92;
                              pos++;
                        }
                  }
                  fName[pos]=0;
                  wr->txChannel[channel]->dataFileName=fName;
                  wr->txChannel[channel]->groupsLoaded=0;
                  wr->txChannel[channel]->groupsTransfered=0;
                  wr->txChannel[channel]->terminateProcess=false;
                  AfxBeginThread(txThread,LPVOID(channel));
            }
            else
            {
                  wr->txChannel[channel]->setDataRate(0);
            }
      }
      for (int channel=0;channel<wr->rxChannelsCount;channel++)
      {
            rxChannelStatus[channel].status-
>SetWindowText("Initializing");
            wr->rxChannel[channel]-
>setFrequency(rxChannelInfo[channel].frequency);
            wr->rxChannel[channel]->k=rxChannelInfo[channel].k;
            wr->rxChannel[channel]-
>dataRate=rxChannelInfo[channel].datarate;
            rxChannelInfo[channel].FIFOInterruptStatus=false;
            rxChannelInfo[channel].FIFOInterruptMask=0;
            char fName[80];
            int pos=0;
            for (int letter=0;
letter<rxChannelInfo[channel].fileName.GetLength(); letter++)
            {
                  fName[pos]=rxChannelInfo[channel].fileName[letter];
                  pos++;
                  if (rxChannelInfo[channel].fileName[letter]==92)
                  {
                        fName[pos]=92;
                        pos++;
                  }
            }
            fName[pos]=0;
            wr->rxChannel[channel]->dataFileName=fName;
            wr->rxChannel[channel]->groupsTransfered=0;
            wr->rxChannel[channel]->groupsSaved=0;
            wr->rxChannel[channel]->terminateProcess=false;
            AfxBeginThread(rxThread,LPVOID(channel));
      }
      //Wait until all threads are ready to transmit or receive
```

```
	unsigned long regBuffer;
	allChannelsReady.ResetEvent();
	::WaitForSingleObject(allChannelsReady, INFINITE);
	//When all channels are ready, enable transmit and receive
	if (wr->rxTxEnable)
	{
		wr->enableRxTx();
	}
	//Loop that checks if there are still active channels
	//or if a stop signal has been issued
	::WaitForSingleObject(allChannelsDone, INFINITE);
	//while (rxTxEnable & (rxThreadsRunning+txThreadsRunning>0));
	//When activity must stop, as a first action stop the card
activity
	wr->disableRxTx();
	//If some channels are still running but a terminate signal has
been issued
	//take care that all channel activity stops
	if(wr->rxThreadsRunning+wr->txThreadsRunning>0)
	{
		for (int channel=0;channel<wr->rxChannelsCount; channel++)
		{
			wr->rxChannel[channel]->terminateProcess=true;
		}
		for (int channel=0;channel<wr->txChannelsCount; channel++)
		{
			wr->txChannel[channel]->terminateProcess=true;
		}
	}
	//Make sure that all channels have finished
	CString displayMessage="Inactive";
	for (int channel=0;channel<wr->rxChannelsCount; channel++)
	{
		while(wr->rxChannel[channel]->threadRunning);
		rxBufferFull[channel].ResetEvent();
	}
	for (int channel=0;channel<wr->txChannelsCount; channel++)
	{
		while(wr->txChannel[channel]->threadRunning);
		txBufferEmpty[channel].ResetEvent();
	}
	//If all channel activity has been terminated but
	//no stop signal has been issued, notify the parent window
	if(wr->rxTxEnable)
	{
		parentWindow->PostMessage(WM_PROCESSES_FINISHED);
	}
	//}
	return error;
}
//*********************************************************
// This is the thread which runs for every RECEPTION channel
//*********************************************************

UINT rxThread(LPVOID pParam)
{
	wr->rxThreadsRunning++;
```

```
	int channel=int(pParam);
	RxChannel* rChannel=wr->rxChannel[channel];
	rChannel->threadRunning=true;
	rChannel->groupsTransfered=0;
	rChannel->groupsSaved=0;
	rChannel->groupCount=1;

	CFile targetFile;
	CString tFile=rChannel->dataFileName.Left(rChannel-
>dataFileName.GetLength()-3)+"TMF";

	int fileOpenError=targetFile.Open(tFile, CFile::modeCreate |
CFile::modeWrite);
	if (fileOpenError==0)
	{
		CString message;
		message.Format("Could not open target file for channel
#%2i.\nChannel will abort.",channel);
		AfxMessageBox(message);
	}
	cSection.Lock();
	wr->threadsReady++;
	if (wr->threadsReady==wr->rxChannelsCount+wr->txChannelsCount)
	{
		allChannelsReady.SetEvent();
	}
	cSection.Unlock();
	rxChannelStatus[channel].status->SetWindowText("Idle...");
	bool rxBufferOverFlow=false;
	if (fileOpenError!=0)
	{
		unsigned short groupCount=1;
		unsigned long rxGroupSize=wr->blockSize*wr-
>rxBlocksPerGroup;
		bool alreadySetStatus=false;
		//Loop to be executed while there is data to save
	  while ((wr->rxTxEnable) && (!rxBufferOverFlow))
		{

	::WaitForSingleObject(rxBufferFull[channel],INFINITE);
			rxBufferFull[channel].ResetEvent();
			if (rChannel->groupsTransfered>rChannel-
>groupsSaved+wr->rxGroupsPerChannel-1)
			{
				rxBufferOverFlow=true;
			}
			if (rxChannelInfo[channel].FIFOInterruptStatus)
			{
				rxBufferOverFlow=true;
		}
			if ((wr->rxTxEnable) && (!rxBufferOverFlow))
			{
				unsigned long* bufferPos=rChannel-
>dataBuffer+(groupCount-1)*rxGroupSize;
				targetFile.Write(bufferPos,4*rxGroupSize);
				rChannel->groupsSaved++;
				groupCount++;
```

```
                        if (groupCount>wr->rxGroupsPerChannel)
groupCount=1;
                        if(!alreadySetStatus)
                        {
                              rxChannelStatus[channel].status-
>SetWindowText("Receiving...");
                              alreadySetStatus=true;
                        }
                  }
            }
      }
      targetFile.Close();
      if ((!rxBufferOverFlow) && (rChannel->k>1))
Demodulate(channel,true);
      //CFile::Remove(tFile);
      rChannel->threadRunning=false;
      cSection.Lock();
      wr->rxThreadsRunning--;
      if (wr->rxThreadsRunning+wr->txThreadsRunning==0)
      {
            allChannelsDone.SetEvent();
      }
      cSection.Unlock();
      CString displayMessage="Inactive";
      if (rxChannelInfo[channel].FIFOInterruptMask)
      {
            switch (rxChannelInfo[channel].FIFOInterruptMask)
            {
                  case 1:
                        displayMessage="FIFO Underflow";
                        break;
                  case 8:
                        displayMessage="FIFO Overflow";
                        break;
                  default:
                        displayMessage="Inactive";
            }
      }
      rxChannelStatus[channel].status->SetWindowText(displayMessage);
      rxChannelStatus[channel].progress->SetPos(0);
      return 0;
}

//************************************************************
// This is the thread which runs for every TRANSMISSION channel
//************************************************************

UINT txThread(LPVOID pParam)
{
      //Acquire parameters
      int channel=int(pParam);
      TxChannel* tChannel=wr->txChannel[channel];
      tChannel->threadRunning=true;
      tChannel->groupsLoaded=0;
      tChannel->groupsTransfered=0;
      tChannel->groupCount=1;
      cSection.Lock();
```

```
	wr->txThreadsRunning++;
	cSection.Unlock();
	//Update status box
	txChannelStatus[channel].status->SetWindowText("Modulating...");
	//Modulate data into I and Q channels
	CString txFileName=Modulate(channel);
	txChannelStatus[channel].progress->SetPos(0);
	bool bufferUnderFlow=false;
	if (wr->rxTxEnable)
	{
		txChannelStatus[channel].status-
>SetWindowText("Transmitting ...");
		CFile txFile;
		txFile.Open(txFileName,CFile::modeRead);
		unsigned long totalGroups=ceil(txFile.GetLength()/(4*wr-
>blockSize*wr->txBlocksPerGroup));

		unsigned int symbolsRead=txFile.Read(tChannel-
>dataBuffer,4*wr->txChannelSize)/4;
		//Initially fill all the channel buffer with data
		if (symbolsRead<wr->txChannelSize)
		{
			for (int symbol=symbolsRead; symbol<wr-
>txChannelSize; symbol++)
			{
				*(tChannel->dataBuffer+symbol)=0;
			}
		}
		cSection.Lock();
		wr->threadsReady++;
		if (wr->threadsReady==wr->rxChannelsCount+wr-
>txChannelsCount)
		{
			allChannelsReady.SetEvent();
		}
		cSection.Unlock();
		tChannel->groupsLoaded=wr->txGroupsPerChannel;
		unsigned short groupCount=1;
		unsigned short txGroupSize=wr->blockSize*wr-
>txBlocksPerGroup;
		//Loop to be executed while there is data to add
		while ((wr->rxTxEnable) &&
(txFile.GetPosition()<txFile.GetLength()) && (!bufferUnderFlow))
		{
			::WaitForSingleObject(txBufferEmpty[channel],
INFINITE);
			txBufferEmpty[channel].ResetEvent();
			if (tChannel->groupsTransfered>tChannel-
>groupsLoaded+wr->txGroupsPerChannel-1)
			{
				bufferUnderFlow=true;
			}
			if (txChannelInfo[channel].FIFOInterruptStatus)
				if (tChannel->groupsTransfered<2)
				{

	txChannelInfo[channel].FIFOInterruptStatus=false;
```

```
	txChannelInfo[channel].FIFOInterruptMask=0;
                        }
                        else
                        {
                              bufferUnderFlow=true;
                  }
                  if ((wr->rxTxEnable) && (!bufferUnderFlow))
                  {
                        unsigned long* bufferPos=tChannel-
>dataBuffer+(groupCount-1)*txGroupSize;
                        if (tChannel->k==1)txFile.SeekToBegin();
                        unsigned long
symbolsRead=txFile.Read(bufferPos,4*txGroupSize)/4;
                        if (symbolsRead<txGroupSize)
                        {
                              for (int symbol=symbolsRead;
symbol<txGroupSize; symbol++)
                              {
                                    *(tChannel->dataBuffer+(groupCount-
1)*txGroupSize+symbol)=0;
                              }
                        }
                        txChannelStatus[channel].progress-
>SetPos(100*tChannel->groupsLoaded/totalGroups);
                        groupCount++;
                        if (groupCount>wr->txGroupsPerChannel)
groupCount=1;
                        tChannel->groupsLoaded++;
                  }
            }
            txFile.Close();
            //CFile::Remove(txFileName);
            //Clearing buffers after all data has been transfered
            for (int group=1; group<=wr->txGroupsPerChannel; group++)
            {
                  if (wr->rxTxEnable)
                  {

	::WaitForSingleObject(txBufferEmpty[channel],INFINITE);
                        txBufferEmpty[channel].ResetEvent();
                  }
                  for (int symbol=0; symbol<txGroupSize; symbol++)
                  {
                        *(tChannel->dataBuffer+(groupCount-
1)*txGroupSize+symbol)=0;
                  }
                  groupCount++;
                  if (groupCount>wr->txGroupsPerChannel) groupCount=1;
            }
            txChannelStatus[channel].progress->SetPos(100);
      }
      cSection.Lock();
      wr->txThreadsRunning--;
      if (wr->rxThreadsRunning+wr->txThreadsRunning==0)
      {
            allChannelsDone.SetEvent();
```

```
	}
	cSection.Unlock();
	CString displayMessage="Inactive";
	if (txChannelInfo[channel].FIFOInterruptMask)
	{
		switch (txChannelInfo[channel].FIFOInterruptMask)
		{
			case 1:
				displayMessage="FIFO Underflow";
				break;
			case 8:
				displayMessage="FIFO Overflow";
				break;
			default:
				displayMessage="Inactive";
		}
	}
    txChannelStatus[channel].status->SetWindowText(displayMessage);
	txChannelStatus[channel].progress->SetPos(0);
	tChannel->threadRunning=false;
	return 0;
}
```

MODEMOD.CPP

```
#include "StdAfx.h"
#include "Math.h"
#include "direct.h"

//Actual Modulation - Demodulation routines declarations
CString mPSK_Modulate(int);
void mPSK_Demodulate(int, bool);

// Modulation - Demodulation routines entry points
// Use these entry points just to select the appropriate
// routines of your code.

CString Modulate(int txChannelNum)
{
	return mPSK_Modulate(txChannelNum);
}

void Demodulate(int rxChannelNum, bool createLog)
{
	mPSK_Demodulate(rxChannelNum, createLog);
}

// M-PSK MODULATION ROUTINE
// This routine takes the data from the transmission file
// and creates the file of symbols

CString mPSK_Modulate(int txChannelNum)
{
	unsigned short Amplitude=wr->maxAmplitude/wr-
>txChannel[txChannelNum]->attenuation;
```

```
      short header[24]={1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,-1,-1,1,1,-
1,1,-1,1};
      short k=wr->txChannel[txChannelNum]->k;
      char M=pow(2,k);
      short symbolsPerPacket=wr->blockSize-32;
      short bytesPerPacket=symbolsPerPacket*k/8;
      char dataBufferIn[wr->blockSize/2];
      int dataBufferOut[wr->blockSize];
      UINT actualBytesRead, actualSymbolsToWrite;
      CFile sourceFile, targetFile;
    int I, Q;
      char sampleBuffer;
      float initialPhase;
      unsigned long nBuffer;
      unsigned short dataMask, bytesToRead, symbolsToWrite;

      CString sFile=wr->txChannel[txChannelNum]->dataFileName;
      CString tFile=sFile.Left(sFile.GetLength()-3)+"TMF";
      targetFile.Open(tFile, CFile::modeCreate | CFile::modeWrite);

      // If test tone selected, simply write a series of I=1 and Q=0
      // and exit
      if (k==1)
      {
            int buffer[9000];
            for (int k=0; k<9000; k++)
                  buffer[k]=Amplitude;
            targetFile.Write(buffer, 36000);
            targetFile.Close();
            txChannelStatus[txChannelNum].progress->SetPos(100);
            return tFile;
      }

      sourceFile.Open(sFile,CFile::modeRead);

      switch (k)
      {
            case 2:
                  bytesToRead=1;
                  symbolsToWrite=4;
                  dataMask=0x3;
                  initialPhase=0;
                  break;
            case 3:
                  bytesToRead=3;
                  symbolsToWrite=8;
                  dataMask=0x7;
                  initialPhase=0;
                  break;
            case 4:
                  bytesToRead=1;
                  symbolsToWrite=2;
                  dataMask=0xF;
                  initialPhase=0;
      }

      //Find number of data packets in the file
```

```
	float
packetNum=float(sourceFile.GetLength())/float(bytesPerPacket);
	unsigned int totalPackets=ceil(packetNum);
	int packet=0;
	// While RxTx is enabled and packets remaining to be modulated
	while ((wr->rxTxEnable) & (packet<totalPackets))
	{
		// Update progress bar
		txChannelStatus[txChannelNum].progress->SetPos(
				100*(packet+1)/totalPackets);
		// Fill data buffer
		actualBytesRead=sourceFile.Read(dataBufferIn,
bytesPerPacket);
		actualSymbolsToWrite=ceil(actualBytesRead*8/k);
		// Write packet header using BPSK modulation
		for (int sample=0; sample<24; sample++)
			dataBufferOut[sample]=Amplitude*header[sample];
		// Write number of samples per packet using QPSK modulation
		for (sample=0; sample<8; sample++)
		{
			sampleBuffer= (actualSymbolsToWrite & (0x3 <<
(2*sample))) >> (2*sample);
			I=Amplitude*cos(2*pi*sampleBuffer/4 + initialPhase);
			Q=Amplitude*sin(2*pi*sampleBuffer/4 + initialPhase);
			dataBufferOut[sample+24]=I|(Q<<16);
		}
		// Write actual data in M-PSK
		unsigned int byteIndexIn=0, symbolIndexOut=0;
		while (byteIndexIn<actualBytesRead)
		{
			//Form the integer
			nBuffer=0;
			for (short byte=0; byte<bytesToRead; byte++)
			{
				if (byteIndexIn<actualBytesRead)
				{

	nBuffer=nBuffer+(dataBufferIn[byteIndexIn] << (8*byte));
					byteIndexIn++;
				}
			}
			//For each symbol in the buffer calculate and store
the I and Q channels
			for (unsigned short symbol=0; symbol<symbolsToWrite;
symbol++)
			{
				sampleBuffer=(nBuffer & (dataMask <<
(k*symbol))) >> (k*symbol);
				I=Amplitude*cos(2*pi*sampleBuffer/M +
initialPhase);
				Q=Amplitude*sin(2*pi*sampleBuffer/M +
initialPhase);
				dataBufferOut[32+symbolIndexOut]=I|(Q<<16);
				symbolIndexOut++;
			}
		}
		// Write buffer to target file
```

```
            targetFile.Write(dataBufferOut,
4*(32+actualSymbolsToWrite));
            packet++;
      }
      sourceFile.Close();
      targetFile.Close();
      txChannelStatus[txChannelNum].progress->SetPos(100);
      return tFile;
}

// M-PSK DEMODULATION ROUTINE
// This routine translates the previouly stored samples
// into symbols and bits

void mPSK_Demodulate(int rxChannelNum, bool createLog)
{
      rxChannelStatus[rxChannelNum].status->SetWindowText("Demodulating
...");
      rxChannelStatus[rxChannelNum].progress->SetPos(0);
      RxChannel* rChannel=wr->rxChannel[rxChannelNum];
      char k=rChannel->k;
      CString tFile=rChannel->dataFileName;
      CString sFile=tFile.Left(tFile.GetLength()-3)+"TMF";

      CFile sourceFile, targetFile;
      CStdioFile logFile;

      sourceFile.Open(sFile, CFile::modeRead);
      targetFile.Open(tFile, CFile::modeCreate | CFile::modeWrite);
      if (createLog)
      {
            CString logFileName;
            logFileName.Format("RxChannel #%1u
log.txt",rxChannelNum+1);
            logFile.Open(logFileName,CFile::modeWrite |
CFile::modeCreate);
            logFile.WriteString("   Packet ##    Found at Phase offset
Synced at  ## symbols\n");
            logFile.WriteString("--------------------------------------
---------------------\n");
      }
      unsigned int totalSamples=sourceFile.GetLength()/4;

      // Set variables
      float M=pow(2, k),
              anglePerSymbol=2*pi/M,
              powerThreshold=50,
              ratioThreshold=11,
              offsetThreshold=4*pi/180;
      unsigned short samplingPoint=2,
                           symbolsPerPacket,
                           samplesPerSymbol=4;// S. O. S. rChannel-
>dataRate
      unsigned int packetNum=0,foundAt=0, syncedAt=0;
      short sampleBuffer[8192][2];
    char symbolBuffer[1024];
      char dataBuffer[1024];
```

```
	int actualSamplesRead;

	// Create Sync Sequence
	short Barker[13]={1,1,1,1,1,-1,-1,1,1,-1,1,-1,1};
	short syncSequence[104];
	for (int i=0; i<13; i++)
		for (int j=0; j<samplesPerSymbol; j++)
			syncSequence[i*samplesPerSymbol+j]=Barker[i];

	unsigned short symbolsToRead, bytesToWrite;
	char dataMask;
	float initialPhase;
	switch (k)
	{
		case 2:
			symbolsToRead=4;
			bytesToWrite=1;
			dataMask=0x3;
			initialPhase=0;
			break;
		case 3:
			symbolsToRead=8;
			bytesToWrite=3;
			dataMask=0x7;
			initialPhase=0;
			break;
		case 4:
			symbolsToRead=2;
			bytesToWrite=1;
			dataMask=0xF;
			initialPhase=0;
	}

	int currentSample=0;
	// For every received packet
	while (currentSample<totalSamples)
	{
		// Find where actual data starts being transmitted,
		// by measuring the average power
		int averagePower=0;
		while  ((averagePower<powerThreshold) &&
(currentSample<totalSamples))
		{
			actualSamplesRead=sourceFile.Read(sampleBuffer,
4*samplesPerSymbol)/4;
			if (actualSamplesRead==0) break;
			averagePower=0;
			for (int i=0;i<actualSamplesRead;i++)
			{
				unsigned int
samplePower=sqrt(pow(sampleBuffer[i][0],2)+pow(sampleBuffer[i][1],2));
				averagePower=averagePower+samplePower;
			}
			averagePower=averagePower/actualSamplesRead;
			currentSample=currentSample+actualSamplesRead;
		}
		if (actualSamplesRead==0) break;
```

```
            // Advance by one symbol
            actualSamplesRead=sourceFile.Read(sampleBuffer,
4*samplesPerSymbol)/4;
            if (actualSamplesRead==0) break;
            currentSample=currentSample+actualSamplesRead;

            //Synchronize in phase
            short phaseHits=0;
            float phaseOffset=0, previousOffset=0, totalOffset=0;
            previousOffset=atan2(sampleBuffer[2*samplesPerSymbol-1][1],
                                 sampleBuffer[2*samplesPerSymbol-
1][0]);
            while (phaseHits<6)
            {
            actualSamplesRead=sourceFile.Read(sampleBuffer,
4*samplesPerSymbol)/4;
                  if (actualSamplesRead==0) break;
                  currentSample=currentSample+actualSamplesRead;
                  phaseOffset=0;
                  for (int i=0; i<actualSamplesRead; i++)

      phaseOffset=phaseOffset+atan2(sampleBuffer[i][1],
sampleBuffer[i][0]);
                  phaseOffset=phaseOffset/actualSamplesRead;
                  if (abs(phaseOffset-previousOffset)<offsetThreshold)
                  {
                        phaseHits++;
                        totalOffset=totalOffset+phaseOffset;
                  }
                  else
                  {
                        phaseHits=0;
                        totalOffset=0;
                  }
                  previousOffset=phaseOffset;
            }
            if (actualSamplesRead==0) break;
            phaseOffset=totalOffset/6;
            packetNum++;
            foundAt=currentSample;

            //Synchronize in time;
            bool syncFound=false;
            int syncPosition;
            actualSamplesRead=sourceFile.Read(sampleBuffer,
13*4*samplesPerSymbol)/4;
            if (actualSamplesRead==0) break;
            double prevCorr=0;
            while ((!syncFound) && (actualSamplesRead>0))
            {
                  double power=0, normalizedCorr=0, corr=0;
                  for (int i=0; i<13*samplesPerSymbol; i++)
                  {
                        int
mag=sqrt(pow(sampleBuffer[i][0],2)+pow(sampleBuffer[i][1],2));
```

```
                        double angle=atan2(sampleBuffer[i][1],
sampleBuffer[i][0])-phaseOffset;
                        corr=corr+mag*cos(angle)*syncSequence[i];
                        power=power+mag;
                  }
                  corr=abs(corr);
                  power=power/13;
                  normalizedCorr=corr/power;
                  if ((normalizedCorr<prevCorr) &&
(power>powerThreshold) && (prevCorr>ratioThreshold))
                  {
                        syncFound=true;
                        syncPosition=currentSample-1;
                        currentSample=syncPosition+13*samplesPerSymbol;
                        sourceFile.Seek(-4,CFile::current);
                  }
                  else
                  {
                        prevCorr=normalizedCorr;
                        currentSample++;
                        for (int i=0; i<13*samplesPerSymbol-1;i++)
                        {
                              sampleBuffer[i][0]=sampleBuffer[i+1][0];
                              sampleBuffer[i][1]=sampleBuffer[i+1][1];
                        }
                        short tmpBuffer[2];
                        actualSamplesRead=sourceFile.Read(tmpBuffer,
4)/4;
                        if (actualSamplesRead==0) break;
                        sampleBuffer[13*samplesPerSymbol-
1][0]=tmpBuffer[0];
                        sampleBuffer[13*samplesPerSymbol-
1][1]=tmpBuffer[1];
                  }
            }
            if (actualSamplesRead==0) break;
            syncedAt=syncPosition;

            // Find the number of symbols per packet
            // Always modulated at QPSK

      actualSamplesRead=sourceFile.Read(sampleBuffer,32*samplesPerSymbo
l)/4;
            if (actualSamplesRead<2*samplesPerSymbol) break;
            currentSample=currentSample+actualSamplesRead;
            symbolsPerPacket=0;
            for (int i=0; i<8; i++)
            {
                  double
phase=atan2(sampleBuffer[i*samplesPerSymbol+samplingPoint][1],

sampleBuffer[i*samplesPerSymbol+samplingPoint][0])-phaseOffset;
                  if (phase<0) phase=phase+2*pi;
                  if (phase>2*pi) phase=phase-2*pi;
                  float decision=2*phase/pi;
                  int iDecision;
                  if (ceil(decision)-decision<0.5)
```

```
                        iDecision=ceil(decision);
                  else
                        iDecision=floor(decision);
                  if (iDecision==4) iDecision=0;
                  symbolsPerPacket=symbolsPerPacket | (iDecision <<
2*i);
            }
            if (createLog)
            {
                  CString logBuffer;
                  logBuffer.Format("%12u%12u%12.2f%12u%12u\n",

      packetNum,foundAt,phaseOffset,syncedAt,symbolsPerPacket);
                  logFile.WriteString(logBuffer);
            }


            // Demodulate the packet
            actualSamplesRead=sourceFile.Read(sampleBuffer,
4*symbolsPerPacket*samplesPerSymbol)/4;
            if (actualSamplesRead<symbolsPerPacket*samplesPerSymbol)
break;
            currentSample=currentSample+actualSamplesRead;
            // Find the symbols
            for (int i=0; i<symbolsPerPacket; i++)
            {
                  double
phase=atan2(sampleBuffer[i*samplesPerSymbol+samplingPoint][1],

sampleBuffer[i*samplesPerSymbol+samplingPoint][0])
                                                -phaseOffset;
                  if (phase<0) phase=phase+2*pi;
                  if (phase>2*pi) phase=phase-2*pi;
                  float decision=(phase-initialPhase)/anglePerSymbol;
                  int iDecision;
                  if (ceil(decision)-decision<0.5)
                        iDecision=ceil(decision);
                  else
                        iDecision=floor(decision);
                  if (iDecision==M) iDecision=0;
                  symbolBuffer[i]=iDecision;
            }
            // Construct the bytes
            UINT byteCount=0, symbolsCount=0;
            while (symbolsCount<symbolsPerPacket)
            {
                  //Form the integer
                  int nBuffer=0;
                  for (short symbol=0; symbol<symbolsToRead; symbol++)
                  {
                        if (symbolsCount<symbolsPerPacket)
                        {
                              nBuffer=nBuffer |
int(symbolBuffer[symbolsCount]<<(k*symbol));
                              symbolsCount++;
                        }
                  }
```

```
                  //Find and store the corresponding bytes
                  for (unsigned short byte=0; byte<bytesToWrite;
byte++)
                  {
                        dataBuffer[byteCount]=(nBuffer & (0xFF <<
(8*byte))) >> (8*byte);
                        byteCount++;
                  }
            }
            targetFile.Write(dataBuffer, byteCount);
      rxChannelStatus[rxChannelNum].progress-
>SetPos(100*currentSample/totalSamples);
      }
      if (createLog) logFile.Close();
      sourceFile.Close();
      targetFile.Close();
}
```

MEMORY_MAP.H

```
//************************************
//* PCI CONFIGURATION SPACE REGISTERS *
//************************************

#define     _RECEIVE_DESTINATION_ADDRESS  0x048

#define     _RECEIVE_DMA_CONTROL                0x04C
#define     _IO_WRITE                                 0x40
#define     _MEMORY_WRITE                             0x70  //
Recommended
#define     _CONFIGURATION_WRITE                0xb0
#define     _DMA_ADDRESS_INCREMENT              0x08
#define     _64_BIT_TRANSFER_ENABLE             0x04  // Must be always
set
#define     _SMART_DMA                                0x01

#define     _TRANSMIT_SOURCE_ADDRESS            0x050

#define     _TRANSMIT_DMA_CONTROL               0x054
#define     _IO_READ                                  0x020
#define     _MEMORY_READ                              0x060 //
Recommended
#define     _CONFIGURATION_READ                       0x0A0
//      _DMA_ADDRESS_INCREMENT                as above
//      _64_BIT_TRANSFER_ENABLE               as above
//      _SMART_DMA                                as above

//***************************
//* PCI MEMORY MAP REGISTERS *
//***************************

#define     _RCV_CHANNELS_3_0_GROUP_COUNT 0x000010
#define     _RCV_CHANNELS_7_4_GROUP_COUNT 0x000014
#define     _TRX_CHANNELS_3_0_GROUP_COUNT 0x000018
#define     _TRX_CHANNELS_7_4_GROUP_COUNT 0x00001C
```

```
#define     _FIRMWARE_VERSION                   0x000020

#define     _BOARD_STATUS                             0x000024
#define     _TX_PLL_STATUS                            0x010
#define     _DISCRETE_INPUT_3                   0x008
#define     _DISCRETE_INPUT_2                   0x004
#define     _DISCRETE_INPUT_1                   0x002
#define     _DISCRETE_INPUT_0                   0x001

#define     _INTERRUPT_STATUS                   0x000028
#define     _GLOBAL_INTERRUPT                   0x80000000
#define     _TX_PROCESSING_COMPLETE             0x04000000
#define     _RX_PROCESSING_COMPLETE             0x02000000
#define     _AUTO_DMA_ABORT_ID                        0x01E00000
#define     _DMA_ABORT_DETECTED                       0x00100000
#define     _TX_AREA_7_COMPLETE                       0x00080000
#define     _TX_AREA_6_COMPLETE                       0x00040000
#define     _TX_AREA_5_COMPLETE                       0x00020000
#define     _TX_AREA_4_COMPLETE                       0x00010000
#define     _TX_AREA_3_COMPLETE                       0x00008000
#define     _TX_AREA_2_COMPLETE                       0x00004000
#define     _TX_AREA_1_COMPLETE                       0x00002000
#define     _TX_AREA_0_COMPLETE                       0x00001000
#define     _RX_AREA_7_COMPLETE                       0x00000800
#define     _RX_AREA_6_COMPLETE                       0x00000400
#define     _RX_AREA_5_COMPLETE                       0x00000200
#define     _RX_AREA_4_COMPLETE                       0x00000100
#define     _RX_AREA_3_COMPLETE                       0x00000080
#define     _RX_AREA_2_COMPLETE                       0x00000040
#define     _RX_AREA_1_COMPLETE                       0x00000020
#define     _RX_AREA_0_COMPLETE                       0x00000010
#define     _TX_DMA_COMPLETE                    0x00000008
#define     _RX_DMA_COMPLETE                    0x00000004
#define     _TX_FIFO_INTERRUPT                        0x00000002
#define     _RX_FIFO_INTERRUPT                        0x00000001

#define     _RECEIVE_FIFO_INTERRUPT_STATUS      0x00002C
#define     _MEMORY_AREA_7                            0xF0000000
#define     _MEMORY_AREA_6                            0x0F000000
#define     _MEMORY_AREA_5                            0x00F00000
#define     _MEMORY_AREA_4                            0x000F0000
#define     _MEMORY_AREA_3                            0x0000F000
#define     _MEMORY_AREA_2                            0x00000F00
#define     _MEMORY_AREA_1                            0x000000F0
#define     _MEMORY_AREA_0                            0x0000000F
#define     _FIFO_UNDERFLOW                           0b0001
#define     _FIFO_EMPTY                               0b0010
#define     _FIFO_EXCEEDS_THRESHOLD             0b0100
#define     _FIFO_OVERFLOW                            0b1000

#define     _TRANSMIT_FIFO_INTERRUPT_STATUS     0x000030
//#define   _MEMORY_AREA_7                            0xF0000000
//#define   _MEMORY_AREA_6                            0x0F000000
//#define   _MEMORY_AREA_5                            0x00F00000
//#define   _MEMORY_AREA_4                            0x000F0000
//#define   _MEMORY_AREA_3                            0x0000F000
//#define   _MEMORY_AREA_2                            0x00000F00
```

```
//#define   _MEMORY_AREA_1                          0x000000F0
//#define   _MEMORY_AREA_0                          0x0000000F
//#define   _FIFO_UNDERFLOW                         0b0001
//#define   _FIFO_EMPTY                             0b0010
//#define   _FIFO_EXCEEDS_THRESHOLD           0b0100
//#define   _FIFO_OVERFLOW                          0b1000

#define _GLOBAL_INTERRUPT_MASK                0x000040
#define _ENABLE_INTERRUPTS                    0x1
#define _DISABLE_INTERRUPTS                   0x0

#define _INTERRUPT_MASK                       0x000044
#define _DMA_ABORT_DETECTED_ENABLE        0x100000
#define _TX_AREA_7_COMPLETE_ENABLE        0x080000
#define _TX_AREA_6_COMPLETE_ENABLE        0x040000
#define _TX_AREA_5_COMPLETE_ENABLE        0x020000
#define _TX_AREA_4_COMPLETE_ENABLE        0x010000
#define _TX_AREA_3_COMPLETE_ENABLE        0x008000
#define _TX_AREA_2_COMPLETE_ENABLE        0x004000
#define _TX_AREA_1_COMPLETE_ENABLE        0x002000
#define _TX_AREA_0_COMPLETE_ENABLE        0x001000
#define _RX_AREA_7_COMPLETE_ENABLE        0x000800
#define _RX_AREA_6_COMPLETE_ENABLE        0x000400
#define _RX_AREA_5_COMPLETE_ENABLE        0x000200
#define _RX_AREA_4_COMPLETE_ENABLE        0x000100
#define _RX_AREA_3_COMPLETE_ENABLE        0x000080
#define _RX_AREA_2_COMPLETE_ENABLE        0x000040
#define _RX_AREA_1_COMPLETE_ENABLE        0x000020
#define _RX_AREA_0_COMPLETE_ENABLE        0x000010
#define _TX_DMA_COMPLETE_ENABLE               0x000008
#define _RX_DMA_COMPLETE_ENABLE               0x000004
#define _TX_INTERRUPTS_DISABLE            0xFFFFFFF5
#define _RX_INTERRUPTS_DISABLE            0xFFFFFFFA
#define     _TX_FIFO_INTERRUPT_ENABLE         0x000002
#define _RX_FIFO_INTERRUPT_ENABLE         0x000001

#define _RECEIVE_FIFO_INTERRUPT_MASK      0x000048
//#define _MEMORY_AREA_7                            0xF0000000
//#define _MEMORY_AREA_6                            0x0F000000
//#define _MEMORY_AREA_5                            0x00F00000
//#define _MEMORY_AREA_4                            0x000F0000
//#define _MEMORY_AREA_3                            0x0000F000
//#define _MEMORY_AREA_2                            0x00000F00
//#define _MEMORY_AREA_1                            0x000000F0
//#define _MEMORY_AREA_0                            0x0000000F
#define     _FIFO_UNDERFLOW_ENABLE            0b0001
#define     _FIFO_EMPTY_ENABLE                    0b0010
#define     _FIFO_EXCEEDS_THRESHOLD_ENABLE    0b0100
#define     _FIFO_OVERFLOW_ENABLE             0b1000

#define _TRANSMIT_FIFO_INTERRUPT_MASK     0x00004C
//#define _MEMORY_AREA_7                            0xF0000000
//#define _MEMORY_AREA_6                            0x0F000000
//#define _MEMORY_AREA_5                            0x00F00000
//#define _MEMORY_AREA_4                            0x000F0000
//#define _MEMORY_AREA_3                            0x0000F000
//#define _MEMORY_AREA_2                            0x00000F00
```

```
//#define _MEMORY_AREA_1                              0x000000F0
//#define _MEMORY_AREA_0                              0x0000000F
//#define _FIFO_UNDERFLOW_ENABLE                0b0001
//#define _FIFO_EMPTY_ENABLE                    0b0010
//#define _FIFO_EXCEEDS_THRESHOLD_ENABLE  0b0100
//#define _FIFO_OVERFLOW_ENABLE                       0b1000

#define _DISCRETE_OUTPUT_CONTROL          0x000050
#define _8_BIT_OUTPUT_MODE_SELECT         0x100
#define _DISCRETE_OUTPUT_7_SELECTED       0x080
#define _DISCRETE_OUTPUT_6_SELECTED       0x040
#define _DISCRETE_OUTPUT_5_SELECTED       0x020
#define _DISCRETE_OUTPUT_4_SELECTED       0x010
#define _DISCRETE_OUTPUT_3_SELECTED       0x008
#define _DISCRETE_OUTPUT_2_SELECTED       0x004
#define _DISCRETE_OUTPUT_1_SELECTED       0x002
#define _DISCRETE_OUTPUT_0_SELECTED       0x001

#define _AUTO_DMA_CONTROL                       0x000094
//31:16 Number of 64-bit words to be transfered
#define _TX_MEMORY_AREA_0_TO_0                  0x0
#define _TX_MEMORY_AREA_0_TO_1                  0x080
#define _TX_MEMORY_AREA_0_TO_2                  0x100
#define _TX_MEMORY_AREA_0_TO_3                  0x180
#define _TX_MEMORY_AREA_0_TO_4                  0x200
#define _TX_MEMORY_AREA_0_TO_5                  0x280
#define _TX_MEMORY_AREA_0_TO_6                  0x300
#define _TX_MEMORY_AREA_0_TO_7                  0x038
#define     _RX_MEMORY_AREA_0_TO_0              0x0
#define _RX_MEMORY_AREA_0_TO_1                  0x008
#define _RX_MEMORY_AREA_0_TO_2                  0x010
#define _RX_MEMORY_AREA_0_TO_3                  0x018
#define _RX_MEMORY_AREA_0_TO_4                  0x020
#define _RX_MEMORY_AREA_0_TO_5                  0x028
#define _RX_MEMORY_AREA_0_TO_6                  0x030
#define _RX_MEMORY_AREA_0_TO_7                  0x038
#define     _AUTO_COUNTERS_RELOAD               0x004 // Allows
multiple transfers automatically
#define _TX_AUTO_DMA_ENABLE                     0x006
#define _RX_AUTO_DMA_ENABLE                     0x005
#define _TX_AUTO_DMA_DISABLE              0xFFFFFFFD
#define _RX_AUTO_DMA_DISABLE              0xFFFFFFFE

// 0x00400:0x0004F0     AUTO DMA BLOCK COUNT
#define _RX_MEMORY_AREA_0_BLOCK_COUNT     0x000400
#define _RX_MEMORY_AREA_1_BLOCK_COUNT     0x000410
#define _RX_MEMORY_AREA_2_BLOCK_COUNT     0x000420
#define _RX_MEMORY_AREA_3_BLOCK_COUNT     0x000430
#define _RX_MEMORY_AREA_4_BLOCK_COUNT     0x000440
#define _RX_MEMORY_AREA_5_BLOCK_COUNT     0x000450
#define _RX_MEMORY_AREA_6_BLOCK_COUNT     0x000460
#define _RX_MEMORY_AREA_7_BLOCK_COUNT     0x000470
#define _TX_MEMORY_AREA_0_BLOCK_COUNT     0x000480
#define _TX_MEMORY_AREA_1_BLOCK_COUNT     0x000490
#define _TX_MEMORY_AREA_2_BLOCK_COUNT     0x0004A0
#define _TX_MEMORY_AREA_3_BLOCK_COUNT     0x0004B0
#define _TX_MEMORY_AREA_4_BLOCK_COUNT     0x0004C0
```

```
#define _TX_MEMORY_AREA_5_BLOCK_COUNT     0x0004D0
#define _TX_MEMORY_AREA_6_BLOCK_COUNT     0x0004E0
#define _TX_MEMORY_AREA_7_BLOCK_COUNT     0x0004F0
// 9:0 Block Count: Number of DMA blocks to be transfered before a DMA
Block Complete int

// 0X000500:0X0005F0 AUTO DMA GROUP COUNT
#define _RX_MEMORY_AREA_0_GROUP_COUNT     0x000500
#define _RX_MEMORY_AREA_1_GROUP_COUNT     0x000510
#define _RX_MEMORY_AREA_2_GROUP_COUNT     0x000520
#define _RX_MEMORY_AREA_3_GROUP_COUNT     0x000530
#define _RX_MEMORY_AREA_4_GROUP_COUNT     0x000540
#define _RX_MEMORY_AREA_5_GROUP_COUNT     0x000550
#define _RX_MEMORY_AREA_6_GROUP_COUNT     0x000560
#define _RX_MEMORY_AREA_7_GROUP_COUNT     0x000570
#define _TX_MEMORY_AREA_0_GROUP_COUNT     0x000580
#define _TX_MEMORY_AREA_1_GROUP_COUNT     0x000590
#define _TX_MEMORY_AREA_2_GROUP_COUNT     0x0005A0
#define _TX_MEMORY_AREA_3_GROUP_COUNT     0x0005B0
#define _TX_MEMORY_AREA_4_GROUP_COUNT     0x0005C0
#define _TX_MEMORY_AREA_5_GROUP_COUNT     0x0005D0
#define _TX_MEMORY_AREA_6_GROUP_COUNT     0x0005E0
#define _TX_MEMORY_AREA_7_GROUP_COUNT     0x0005F0
// 4:0 Block Count: Number of DMA groups to be transfered before the
initial
//                  DMA address is reloaded


// 0X000800:0X0008F0 AUTO DMA ADDRESS
#define _RX_MEMORY_AREA_0_ADDRESS         0x000800
#define _RX_MEMORY_AREA_1_ADDRESS         0x000810
#define _RX_MEMORY_AREA_2_ADDRESS         0x000820
#define _RX_MEMORY_AREA_3_ADDRESS         0x000830
#define _RX_MEMORY_AREA_4_ADDRESS         0x000840
#define _RX_MEMORY_AREA_5_ADDRESS         0x000850
#define _RX_MEMORY_AREA_6_ADDRESS         0x000860
#define _RX_MEMORY_AREA_7_ADDRESS         0x000870
#define _TX_MEMORY_AREA_0_ADDRESS         0x000880
#define _TX_MEMORY_AREA_1_ADDRESS         0x000890
#define _TX_MEMORY_AREA_2_ADDRESS         0x0008A0
#define _TX_MEMORY_AREA_3_ADDRESS         0x0008B0
#define _TX_MEMORY_AREA_4_ADDRESS         0x0008C0
#define _TX_MEMORY_AREA_5_ADDRESS         0x0008D0
#define _TX_MEMORY_AREA_6_ADDRESS         0x0008E0
#define _TX_MEMORY_AREA_7_ADDRESS         0x0008F0
// 31:0 Starting address in memory to begin DMA transfer

#define     _RECEIVE_CONTROL                    0x001100
//        _RX_CHANNEL_ORGANIZATION
#define     _8_CHANNELS                               0x0000
#define     _2_POLYPHASE_CHANNELS               0x4000
#define     _4_POLYPHASE_CHANNELS               0x8000
#define     _8_POLYPHASE_CHANNELS               0xC000
#define     _BIT_REGISTERS_ENABLE               0x2000
#define     _FIFO_FLUSH                               0x1000
#define     _RX_HEADER_ENABLE                   0x0400
```

```
#define     _RX_MASTER_ENABLE                   0x0200      // Must be
selected
#define     _POLYPHSE_SEQUENCIAL_DATA           0x0100      // Usually
not set
#define     _1_RX_MEMORY_AREA                   0x0
#define     _2_RX_MEMORY_AREAS                        0x0010
#define     _3_RX_MEMORY_AREAS                        0x0020
#define     _4_RX_MEMORY_AREAS                        0x0030
#define     _5_RX_MEMORY_AREAS                        0x0040
#define     _6_RX_MEMORY_AREAS                        0x0050
#define     _7_RX_MEMORY_AREAS                        0x0060
#define     _8_RX_MEMORY_AREAS                        0x0070
#define     _RX_CIRCUITRY_ENABLE                0x0002      // Enables
receiver circuitry
#define     _RX_ENABLE                                0x0001      //
Should be enabled after initialization
#define _RX_DISABLE                                   0xFFFFFFFE

#define     _MANUAL_DMA_RX_MEMORY_SELECT  0x001104
#define     _MANUAL_RX_MEMORY_SELECT            0x100
#define     _START_MANUAL_DMA_RX                0x010
#define _RX_DIRECT_FIFO_ACCESS                  0x008
#define     _RX_MEMORY_AREA_SELECT              0X007
// Ox00X = Memory area to be used for transfer (0-7)

#define     _RECEIVE_TIMING_CONTROL             0x001108
#define     _TIMING_CONTROL_DISABLE             0x0

#define     _RECEIVE_CLOCK_RATE                       0x001128    //For
93MHz=92999998

//Registers of pages 67-73 are not used.

#define     _RX_MEMORY_AREA_0_ORGANIZATION      0x001300
#define     _RX_MEMORY_AREA_1_ORGANIZATION      0x001310
#define     _RX_MEMORY_AREA_2_ORGANIZATION      0x001320
#define     _RX_MEMORY_AREA_3_ORGANIZATION      0x001330
#define     _RX_MEMORY_AREA_4_ORGANIZATION      0x001340
#define     _RX_MEMORY_AREA_5_ORGANIZATION      0x001350
#define     _RX_MEMORY_AREA_6_ORGANIZATION      0x001360
#define     _RX_MEMORY_AREA_7_ORGANIZATION      0x001370

//For the above registers, the following fields are used
#define _RX_END_CHANNEL_0                       0x0
#define _RX_END_CHANNEL_1                       0x1000
#define _RX_END_CHANNEL_2                       0x2000
#define _RX_END_CHANNEL_3                       0x3000
#define _RX_END_CHANNEL_4                       0x4000
#define _RX_END_CHANNEL_5                       0x5000
#define _RX_END_CHANNEL_6                       0x6000
#define _RX_END_CHANNEL_7                       0x7000
#define _RX_START_CHANNEL_0                     0x0
#define _RX_START_CHANNEL_1                     0x0100
#define _RX_START_CHANNEL_2                     0x0200
#define _RX_START_CHANNEL_3                     0x0300
#define _RX_START_CHANNEL_4                     0x0400
#define _RX_START_CHANNEL_5                     0x0500
```

```
#define _RX_START_CHANNEL_6                    0x0600
#define _RX_START_CHANNEL_7                    0x0700
#define     _RX_CHANNEL_0_DIRECTED             0x0001
#define     _RX_CHANNEL_1_DIRECTED             0x0002
#define     _RX_CHANNEL_2_DIRECTED             0x0004
#define     _RX_CHANNEL_3_DIRECTED             0x0008
#define     _RX_CHANNEL_4_DIRECTED             0x0010
#define     _RX_CHANNEL_5_DIRECTED             0x0020
#define     _RX_CHANNEL_6_DIRECTED             0x0040
#define     _RX_CHANNEL_7_DIRECTED             0x0080

#define _RX_MEMORY_AREA_0_POINTER         0x001400
#define _RX_MEMORY_AREA_1_POINTER         0x001410
#define _RX_MEMORY_AREA_2_POINTER         0x001420
#define _RX_MEMORY_AREA_3_POINTER         0x001430
#define _RX_MEMORY_AREA_4_POINTER         0x001440
#define _RX_MEMORY_AREA_5_POINTER         0x001450
#define _RX_MEMORY_AREA_6_POINTER         0x001460
#define _RX_MEMORY_AREA_7_POINTER         0x001470

//For the above registers, the following values must be entered:
// 30:16    Last address of the designated memory area in relation
//          to the starting address of the Rx memory block (64-bit
longwords)
// 0:14           The first address of the designated memory area (as
above)

#define _RX_MEMORY_AREA_0_LIMITS          0x001500
#define _RX_MEMORY_AREA_1_LIMITS          0x001510
#define _RX_MEMORY_AREA_2_LIMITS          0x001520
#define _RX_MEMORY_AREA_3_LIMITS          0x001530
#define _RX_MEMORY_AREA_4_LIMITS          0x001540
#define _RX_MEMORY_AREA_5_LIMITS          0x001550
#define _RX_MEMORY_AREA_6_LIMITS          0x001560
#define _RX_MEMORY_AREA_7_LIMITS          0x001570

//For the above registers, the following values must be entered:
// 30:16    Size of the designated memory area (64-bit longwords)
// 0:14           Threshold at which interrupt will be generated

#define _RX_MEMORY_AREA_0_PTR_STATUS           0x001800
#define _RX_MEMORY_AREA_1_PTR_STATUS           0x001810
#define _RX_MEMORY_AREA_2_PTR_STATUS           0x001820
#define _RX_MEMORY_AREA_3_PTR_STATUS           0x001830
#define _RX_MEMORY_AREA_4_PTR_STATUS           0x001840
#define _RX_MEMORY_AREA_5_PTR_STATUS           0x001850
#define _RX_MEMORY_AREA_6_PTR_STATUS           0x001860
#define _RX_MEMORY_AREA_7_PTR_STATUS           0x001870

//For the above registers, the following values must be entered:
// 0:14           Next address to be read from the designated FIFO
Memory area
// PROVIDED FOR DEBUG PURPOSES

#define     _RECEIVE_FIFO__STATUS              0x001900
//#define   _MEMORY_AREA_7                           0xF0000000
//#define   _MEMORY_AREA_6                           0x0F000000
```

```
//#define   _MEMORY_AREA_5                          0x00F00000
//#define   _MEMORY_AREA_4                          0x000F0000
//#define   _MEMORY_AREA_3                          0x0000F000
//#define   _MEMORY_AREA_2                          0x00000F00
//#define   _MEMORY_AREA_1                          0x000000F0
//#define   _MEMORY_AREA_0                          0x0000000F
//#define   _FIFO_UNDERFLOW                         0b0001
//#define   _FIFO_EMPTY                             0b0010
//#define   _FIFO_EXCEEDS_THRESHOLD           0b0100
//#define   _FIFO_OVERFLOW                          0b1000

#define _RX_MEMORY_AREA_0_WR_PTR_STATUS         0x001A00
#define _RX_MEMORY_AREA_1_WR_PTR_STATUS         0x001A10
#define _RX_MEMORY_AREA_2_WR_PTR_STATUS         0x001A20
#define _RX_MEMORY_AREA_3_WR_PTR_STATUS         0x001A30
#define _RX_MEMORY_AREA_4_WR_PTR_STATUS         0x001A40
#define _RX_MEMORY_AREA_5_WR_PTR_STATUS         0x001A50
#define _RX_MEMORY_AREA_6_WR_PTR_STATUS         0x001A60
#define _RX_MEMORY_AREA_7_WR_PTR_STATUS         0x001A70

//For the above registers, the following values must be entered:
// 0:14           Next address of the RX Memory to be written.
// PROVIDED FOR DEBUG PURPOSES

#define _RX_MEMORY_AREA_0_FIFO_COUNT            0x001B00
#define _RX_MEMORY_AREA_1_FIFO_COUNT            0x001B10
#define _RX_MEMORY_AREA_2_FIFO_COUNT            0x001B20
#define _RX_MEMORY_AREA_3_FIFO_COUNT            0x001B30
#define _RX_MEMORY_AREA_4_FIFO_COUNT            0x001B40
#define _RX_MEMORY_AREA_5_FIFO_COUNT            0x001B50
#define _RX_MEMORY_AREA_6_FIFO_COUNT            0x001B60
#define _RX_MEMORY_AREA_7_FIFO_COUNT            0x001B70

//For the above registers, the following values must be entered:
// 0:14           Amount of data remaining in the designated Memory
Area FIFO

#define     _TRANSMIT_CONTROL                   0x002100
#define     _TX_BIT_REGISTERS_ENABLE            0x2000
#define     _TX_FIFO_FLUSH                            0x1000
#define _TX_FIFO_ENABLE                         0xFFFFEFFF
#define _TX_MASTER_SYNC_ENABLE                  0x0400      // Must be
set???
#define     _TX_MASTER_ENABLE                   0x0200      // Must be
selected
#define     _1_TX_MEMORY_AREA                   0x0
#define     _2_TX_MEMORY_AREAS                        0x0010
#define     _3_TX_MEMORY_AREAS                        0x0020
#define     _4_TX_MEMORY_AREAS                        0x0030
#define     _5_TX_MEMORY_AREAS                        0x0040
#define     _6_TX_MEMORY_AREAS                        0x0050
#define     _7_TX_MEMORY_AREAS                        0x0060
#define     _8_TX_MEMORY_AREAS                        0x0070
#define     _TX_CIRCUITRY_ENABLE                0x0002      // Enables
transmitter circuitry
#define     _TX_ENABLE                                0x0001      //
Should be enabled after initialization
```

```
#define _TX_DISABLE                                    0xFFFFFFFE

#define     _MANUAL_DMA_TX_MEMORY_SELECT  0x002104
#define     _TX_ENDIAN_SELECT                   0x100
#define     _DIRECT_TX_MEMORY_ACCESS            0x008
#define     _TX_MEMORY_AREA_SELECT              0x007
// 0x00X = Memory area to be used for transfer (0-7)

#define     _TX_TIMING_CONTROL                        0x002108
#define     _TX_TIMING_CONTROL_DISABLE          0x0

#define     _TX_CLOCK_RATE                            0x002128    //For
93MHz=92999998

#define _PRN_CONTROL                            0x00211C
//18:4 PRN Seed value
#define     _4_SAMPLES_PER_SYMBOL               0x8
#define     _2_SAMPLES_PER_SYMBOL               0x4
#define     _1_SAMPLE_PER_SYMBOL                0x0
#define     _PRN_CODE_TX_ENABLE                       0x1
      //If 0 FIFO data is transmitted

#define     _PRN_ZERO_IQ_VALUE                        0x002120
//31:16 Q
//00:15 I

#define _PRN_ONE_IQ_VALUE                       0x002124
//31:16     Q
//00:15 I

//Registers of pages 84-88 are not used.

#define     _TX_MEMORY_AREA_0_ORGANIZATION      0x002300
#define     _TX_MEMORY_AREA_1_ORGANIZATION      0x002310
#define     _TX_MEMORY_AREA_2_ORGANIZATION      0x002320
#define     _TX_MEMORY_AREA_3_ORGANIZATION      0x002330
#define     _TX_MEMORY_AREA_4_ORGANIZATION      0x002340
#define     _TX_MEMORY_AREA_5_ORGANIZATION      0x002350
#define     _TX_MEMORY_AREA_6_ORGANIZATION      0x002360
#define     _TX_MEMORY_AREA_7_ORGANIZATION      0x002370

//For the above registers, the following fields are used
#define _TX_END_CHANNEL_0                       0x0
#define _TX_END_CHANNEL_1                       0x1000
#define _TX_END_CHANNEL_2                       0x2000
#define _TX_END_CHANNEL_3                       0x3000
#define _TX_END_CHANNEL_4                       0x4000
#define _TX_END_CHANNEL_5                       0x5000
#define _TX_END_CHANNEL_6                       0x6000
#define _TX_END_CHANNEL_7                       0x7000
#define _TX_START_CHANNEL_0                     0x0
#define _TX_START_CHANNEL_1                     0x0100
#define _TX_START_CHANNEL_2                     0x0200
#define _TX_START_CHANNEL_3                     0x0300
#define _TX_START_CHANNEL_4                     0x0400
#define _TX_START_CHANNEL_5                     0x0500
#define _TX_START_CHANNEL_6                     0x0600
```

```
#define _TX_START_CHANNEL_7                     0x0700
//The values below can be ORed to allow multiple channels
#define     _TX_CHANNEL_0_DIRECTED              0x0001
#define     _TX_CHANNEL_1_DIRECTED              0x0002
#define     _TX_CHANNEL_2_DIRECTED              0x0004
#define     _TX_CHANNEL_3_DIRECTED              0x0008
#define     _TX_CHANNEL_4_DIRECTED              0x0010
#define     _TX_CHANNEL_5_DIRECTED              0x0020
#define     _TX_CHANNEL_6_DIRECTED              0x0040
#define     _TX_CHANNEL_7_DIRECTED              0x0080

#define _TX_MEMORY_AREA_0_POINTER         0x002400
#define _TX_MEMORY_AREA_1_POINTER         0x002410
#define _TX_MEMORY_AREA_2_POINTER         0x002420
#define _TX_MEMORY_AREA_3_POINTER         0x002430
#define _TX_MEMORY_AREA_4_POINTER         0x002440
#define _TX_MEMORY_AREA_5_POINTER         0x002450
#define _TX_MEMORY_AREA_6_POINTER         0x002460
#define _TX_MEMORY_AREA_7_POINTER         0x002470

//For the above registers, the following values must be entered:
// 30:16    Last address of the designated memory area in relation
//          to the starting address of the Rx memory block (64-bit
longwords)
// 0:14           The first address of the designated memory area (as
above)

#define _TX_MEMORY_AREA_0_LIMITS          0x002500
#define _TX_MEMORY_AREA_1_LIMITS          0x002510
#define _TX_MEMORY_AREA_2_LIMITS          0x002520
#define _TX_MEMORY_AREA_3_LIMITS          0x002530
#define _TX_MEMORY_AREA_4_LIMITS          0x002540
#define _TX_MEMORY_AREA_5_LIMITS          0x002550
#define _TX_MEMORY_AREA_6_LIMITS          0x002560
#define _TX_MEMORY_AREA_7_LIMITS          0x002570

//For the above registers, the following values must be entered:
// 30:16    Size of the designated memory area (64-bit longwords)
// 0:14           Threshold at which interrupt will be generated

#define _TX_MEMORY_AREA_0_PTR_STATUS      0x002800
#define _TX_MEMORY_AREA_1_PTR_STATUS      0x002810
#define _TX_MEMORY_AREA_2_PTR_STATUS      0x002820
#define _TX_MEMORY_AREA_3_PTR_STATUS      0x002830
#define _TX_MEMORY_AREA_4_PTR_STATUS      0x002840
#define _TX_MEMORY_AREA_5_PTR_STATUS      0x002850
#define _TX_MEMORY_AREA_6_PTR_STATUS      0x002860
#define _TX_MEMORY_AREA_7_PTR_STATUS      0x002870

//For the above registers, the following values must be entered:
// 0:14           Next address to be read from the designated FIFO
Memory area
// PROVIDED FOR DEBUG PURPOSES

#define     _TRANSMIT_FIFO__STATUS              0x002900
//#define   _MEMORY_AREA_7                            0xF0000000
//#define   _MEMORY_AREA_6                            0x0F000000
```

```
//#define   _MEMORY_AREA_5                          0x00F00000
//#define   _MEMORY_AREA_4                          0x000F0000
//#define   _MEMORY_AREA_3                          0x0000F000
//#define   _MEMORY_AREA_2                          0x00000F00
//#define   _MEMORY_AREA_1                          0x000000F0
//#define   _MEMORY_AREA_0                          0x0000000F
//#define   _FIFO_UNDERFLOW                         0b0001
//#define   _FIFO_EMPTY                             0b0010
//#define   _FIFO_EXCEEDS_THRESHOLD           0b0100
//#define   _FIFO_OVERFLOW                          0b1000

#define _TX_MEMORY_AREA_0_RD_PTR_STATUS         0x002A00
#define _TX_MEMORY_AREA_1_RD_PTR_STATUS         0x002A10
#define _TX_MEMORY_AREA_2_RD_PTR_STATUS         0x002A20
#define _TX_MEMORY_AREA_3_RD_PTR_STATUS         0x002A30
#define _TX_MEMORY_AREA_4_RD_PTR_STATUS         0x002A40
#define _TX_MEMORY_AREA_5_RD_PTR_STATUS         0x002A50
#define _TX_MEMORY_AREA_6_RD_PTR_STATUS         0x002A60
#define _TX_MEMORY_AREA_7_RD_PTR_STATUS         0x002A70

//For the above registers, the following values must be entered:
// 0:14           Next address of the TX Memory to be READ.
// PROVIDED FOR DEBUG PURPOSES

#define _TX_MEMORY_AREA_0_FIFO_COUNT            0x002B00
#define _TX_MEMORY_AREA_1_FIFO_COUNT            0x002B10
#define _TX_MEMORY_AREA_2_FIFO_COUNT            0x002B20
#define _TX_MEMORY_AREA_3_FIFO_COUNT            0x002B30
#define _TX_MEMORY_AREA_4_FIFO_COUNT            0x002B40
#define _TX_MEMORY_AREA_5_FIFO_COUNT            0x002B50
#define _TX_MEMORY_AREA_6_FIFO_COUNT            0x002B60
#define _TX_MEMORY_AREA_7_FIFO_COUNT            0x002B70

//For the above registers, the following values must be entered:
// 0:14           Amount of data remaining in the designated Memory
Area FIFO

#define _DITHER_NOISE_POWER_CONTROL             0x008000
//7:0 Noise power from -90dBm (0x0) to -30 dBm (0xFF)

#define _ATTENUATOR_POWER_CONTROL               0x008004

//Registers of pages 95-100 are not used
```

PMCRADIOI.H

```
//Red River Engineering
//Include file for use with WaveRunner Multi-Card Library
// PN:  SRC-905-008-R00 (August 16, 2001)
// Author - Patrick Jennings

#ifndef PMCRADIOI__H
#define PMCRADIOI__H

#ifdef __cplusplus
```

```
extern "C" {
#endif

typedef struct {
      int NumBuffers;
      unsigned long BufSizeBytes;
      unsigned long ErrStatus;
      unsigned long v_dmap[32];
      unsigned long p_dmap[32];
}     s_DMAConf;

typedef struct {
      int DevNum;       //Device Number of the Radio
} s_PMCRadio;

/*
s_PMCRadio Radio0;
Radio0.DevNum = 0;

OpenPMCRadio(&Radio0);
ClosePMCRadio(&Radio0);
*/

////////////////////////////////////////////////////////////////////////
///////
//                                    PROTYPES
////////////////////////////////////////////////////////////////////////
///////
/*====================================================================
=================
Prototype   int OpenWaveRunner(void);

Function          OpenWaveRunner instantiates a Wave Walker radio as a
Windows device and
memory maps its physical PCI space to a local memory image.  The local
memory image is
used to access the radio via the Wave Walker memory map described in
the Wave Walker Hardware Reference Manual.
OpenWaveRunner must be called before any of the Wave Walker library
functions can be used.
CloseWaveRunner must be called prior to exiting a program that has
called OpenWaveRunner
to prevent memory leaks.

Return Values
0 Successful open and memory mapping of Wave Walker Radio
-1    Wave Walker device not accessible
-2    OS unable to memory map device
====================================================================
===============*/
int OpenWaveRunner();                                   //Opens a
WaveRunner and maps it
int OpenMultiWaveRunner(int iDeviceNum);
int OpenPMCRadio(s_PMCRadio * PMCRadio);

/*====================================================================
=================
```

```
Prototype   int CloseWaveRunner(void);

Function          Unmaps and deallocates system memory assigned to a
Wave walker device.
This function must be called prior to exiting any program that has
issued an
OpenWaveRunner command.  Wave Walker library functions can no longer be
used after
this function is executed unless a new OpenWaveRunner command is
issued.

Return Values
0 Successful operation
-1    Unable to deallocate memory (system fault)
================================================================
===============*/
int CloseWaveRunner();                                  //Closes and
deletes a WaveRunner
int CloseMultiWaveRunner(int iDeviceNum);
int ClosePMCRadio(s_PMCRadio * PMCRadio);

/*============================================================================
=================
Prototype   int GetMemPointer(unsigned long *Memptr);

Function          GetMemPointer returns a pointer to the Wave Walker
local memory image.  This pointer can be used to access any location in
the Wave Walker memory map.  The user must pass the pointer by
reference by calling as follows:

unsigned long  *MyWaveRunnerptr;           //User defined pointer to
Wave Walker
      GetMemPointer((unsigned long *) &MyWaveRunnerptr);     //Call to
initialize pointer

Once initialized the pointer can be used to access Wave Walker as
follows:

Write to the transmitter real data FIFO:

#define     TX_REAL_OFFSET    0x1000
*(MyWaveRunnerptr + TX_REAL_OFFSET/4) = 0x12345678;

Read from Wave Walker status register:

#define     STATUS_REG_OFFSET 0x058C
      data = *(MyWaveRunnerptr + STATUS_REG_OFFSET/4);

note: Wave Walker memory map addresses must be divided by 4 when using
the pointer
returned by GetMemPointer.  The division by 4 translates the byte
offset memory map
listings to 32-bit (4 byte) word pointer increments.  This is NOT
required for the
Write and Read Wave Walker functions found elsewhere in this library.

Return Values
```

```
0     Successful operation
-1    Unsuccessful operation
================================================================================*/
int GetMemPointer(unsigned long* ptr);          //Provides a pointer to
the virtual map of the radio
int GetMultiMemPointer(int iDeviceNum, unsigned long* ptr);
int GetPMCRadioMemPointer(s_PMCRadio * , unsigned long* ptr);

/*================================================================================
=================
Prototype   int WriteWaveRunner(unsigned long AddressOffset, unsigned
long Data);

Function          The WriteWaveRunner function writes the data value
passed in
variable Data to the Wave Walker device memory mapped register
indicated by
variable AddressOffset.  AddressOffset is any valid Wave Walker memory
map address.
Please see the Wave Walker Hardware Reference Manual for a listing of
valid memory
map offset addresses.

Return Values
0     Successful operation
-1    Unsuccessful operation
================================================================================
===============*/
int WriteWaveRunner(unsigned long AddressOffset, unsigned long Data);
//Writes
int WriteMultiWaveRunner(int iDeviceNum, unsigned long AddressOffset,
unsigned long Data);
int WritePMCRadio(s_PMCRadio * PMCRadio, unsigned long AddressOffset,
unsigned long Data); //Writes


/*================================================================================
=================
Prototype   int ReadWaveRunner(unsigned long AddressOffset, unsigned
long *Data);

Function          ReadWaveRunner returns the value located at
AddressOffset to the variable Data passed by reference. AddressOffset
is any valid Wave Walker memory map address.

Calling form:

            unsigned long  address, data;        // User defined
variables
            ReadWaveRunner(address, &data);           // Call to read
data

Data value returned by reference.

Return Values
0     Successful operation
```

```
-1    Unsuccessful operation

==========================================================================
===============*/
int ReadWaveRunner(unsigned long AddressOffset, unsigned long *Data);
//Reads by ref
int ReadMultiWaveRunner(int iDeviceNum, unsigned long AddressOffset,
unsigned long *Data);
int ReadPMCRadio(s_PMCRadio * PMCRadio, unsigned long AddressOffset,
unsigned long *Data); //Reads by ref




/*==========================================================================
=============
int ReadWRConfigSpace(unsigned long nOffset, char *PCIconfig, unsigned
long nBytes);
int ReadMultiWRConfigSpace(int iDeviceNum, unsigned long nOffset, char
*PCIconfig, unsigned long nBytes)

==========================================================================
===========*/
int ReadWRConfigSpace(unsigned long nOffset, char *PCIconfig, unsigned
long nBytes);
int ReadMultiWRConfigSpace(int iDeviceNum, unsigned long nOffset, char
*PCIconfig, unsigned long nBytes);
int ReadPMCRadioConfigSpace(s_PMCRadio * PMCRadio,unsigned long
nOffset, char *PCIconfig, unsigned long nBytes);


/*==========================================================================
=============
int WriteWRConfigSpace(unsigned long nOffset, char *PCIconfig, unsigned
long nBytes)
int WriteMultiWRConfigSpace(int iDeviceNum, unsigned long nOffset, char
*PCIconfig, unsigned long nBytes)

==========================================================================
===========*/
int WriteWRConfigSpace(unsigned long nOffset, char *PCIconfig, unsigned
long nBytes);
int WriteMultiWRConfigSpace(int iDeviceNum, unsigned long nOffset, char
*PCIconfig, unsigned long nBytes);
int WritePMCRadioConfigSpace(s_PMCRadio * PMCRadio, unsigned long
nOffset, char *PCIconfig, unsigned long nBytes);

/*==========================================================================
=================
Prototype   int GetFirmRev(unsigned long *date);

Function          GetFirmRev returns the contents of the Wave Walker
firmware revision
register.  A read of the firmware revision register can be used to
quickly verify
```

```
the communication path to the Wave Walker radio.  The firmware revision
date is a
hexadecimal value passed by reference.

Calling form:

      unsigned long           date;       // User defined variable for
the revision date

      GetFirmRev(&date);            // Function call to write firmware
revision date

The variable date is updated with the contents of the Wave Walker
firmware revision
register (32 bit Hex constant, for example 0x09171999, note the date
only makes sense
when viewed as an unsigned long hexadecimal number)  A date of all 0's
indicates an
access problem.

Return Values
0     Successful operation
-1    Unsuccessful operation

================================================================================*/

int GetFirmRev(unsigned long * date);                        //Shows the
Firmware revison
int GetMultiFirmRev(int iDeviceNum, unsigned long * date);
int GetPMCRadioFirmRev(s_PMCRadio * PMCRadio, unsigned long * date);



/*================================================================================
Prototype   int ConfigWaveRunner(char ConfigFname [80]);

Function          This function is used to load Wave Walker
configuration files created
using the Waveformer configuration tool.  The Waveformer tool creates
two sets of three
configuration files.  One set has a ".h" extension, the other has a
".txt" extension.
The ConfigWaveRunner function indirectly uses the ".h" versions of
these files.
Indirectly means that the filename passed to the function as
ConfigFname contains a list
of the .h files to be uploaded.  For example consider a file named
"ConfigExample.txt".
 The file is a text file with three entries as follows:

      bdinit.h
      txinit.h
      rxinit.h

(note the User may modify the file names)
```

```
The calling sequence for ConfigWaveRunner using the example file is:

      ConfigWaveRunner("ConfigExample.txt");

The function opens the pointer file "ConfigExample.txt" to find the
three configuration
file names it contains.  The contents of the three .h files are
automatically uploaded
to the Wave Walker radio.

Note:  The configuration pointer file and its ".h" files must all be
collocated in a
working directory or path recognized by the User application program.

Return Values
0     Successful operation
-1    Unable to find configuration pointer file
-2    Unable to find one or more of the ".h" files
-3    System unable to communicate with Radio

=================================================================
===============*/
int ConfigWaveRunner (char ConfigFname [80]); //Configs a WaveRunner
from the Excel tool
int ConfigMultiWaveRunner (int iDeviceNum, char ConfigFname [80]);
int ConfigPMCRadio (s_PMCRadio * PMCRadio, char ConfigFname [80]);
//Configs a WaveRunner from the Excel tool

/*=================================================================
===============*/
// void WaveRunnerIsr(unsigned long status)
// WaveRunnerIsr is the entry point for any interrupt generated by a
WaveRunner
// device.  This function must always be included in any program that
uses the
// Wave Walker windows library.  Uncomment and move this function into
your main code space
// and replace the printf statement with your own ISR code if you are
using interrupts.
/*=================================================================
===============*/

int GetDMAPA(unsigned long *wrdmapa, unsigned long *wrdmava);
int GetMultiDMAPA(int iDeviceNum, unsigned long *wrdmapa, unsigned long
*wrdmava);
int GetPMCRadioDMAPA(s_PMCRadio * PMCRadio, unsigned long *wrdmapa,
unsigned long *wrdmava);

/*=================================================================
===============*/
// unsigned long ReturnMaxDMABufferSize()
//
// Returns the size in bytes allocated for each M301 to do DMA
transfers
/*=================================================================
===============*/
```

```
unsigned long GetMaxDMABufferSize();
unsigned long GetMultiMaxDMABufferSize(int iDeviceNum);
unsigned long GetPMCRadioMaxDMABufferSize(s_PMCRadio * PMCRadio);

void SetupDMABuffers(int iDevNum, s_DMAConf * DMAConfig);
void SetupPMCRadioDMABuffers(s_PMCRadio * PMCRadio, s_DMAConf *
DMAConfig);

void CloseDMABuffers(int iDevNum, s_DMAConf * DMAConfig);
void ClosePMCRadioBuffers(s_PMCRadio * PMCRadio, s_DMAConf *
DMAConfig);

void SetDMABufferSize(int Pages);
void SetMultiDMABufferSize(int iDeviceNum, int Pages);
void SetPMCRadioDMABufferSize(s_PMCRadio * PMCRadio,int Pages);

int Count301s(void);
int CountPMCRadios(void);

void PMCRadioIsr(unsigned long status);
void PMCRadioIsr0(unsigned long status);
void PMCRadioIsr1(unsigned long status);
void PMCRadioIsr2(unsigned long status);
void PMCRadioIsr3(unsigned long status);
void PMCRadioIsr4(unsigned long status);
void PMCRadioIsr5(unsigned long status);
void PMCRadioIsr6(unsigned long status);
void PMCRadioIsr7(unsigned long status);

#ifndef NOISR
/*
Move the following lines to your program if you are going
to use interrupts.  Replace the printf statement with your ISR code.
*/

#ifndef USERISR
void WaveRunnerIsr(unsigned long status)
{
      PMCRadioIsr0(status);
}
#endif

#ifndef USERISR0
void PMCRadioIsr0(unsigned long status)
{
}
#endif

#ifndef USERISR1
void PMCRadioIsr1(unsigned long status)
{
}
#endif

#ifndef USERISR2
void PMCRadioIsr2(unsigned long status)
{
```

```
}
#endif

#ifndef USERISR3
void PMCRadioIsr3(unsigned long status)
{
}
#endif

#ifndef USERISR4
void PMCRadioIsr4(unsigned long status)
{
}
#endif

#ifndef USERISR5
void PMCRadioIsr5(unsigned long status)
{
}
#endif

#ifndef USERISR6
void PMCRadioIsr6(unsigned long status)
{
}
#endif

#ifndef USERISR7
void PMCRadioIsr7(unsigned long status)
{
}
#endif

#endif

char * QueryRRProductID(void);
char * QueryLibBuildDateString(void);
unsigned long QueryDriverXLibVersion(void);
unsigned long QueryLibBuildDate(void);

/*
start sync and end sync are used to coordinate global variable use
outside the ISR routine, a start sync and end sync should frame
any statement(s) that use(s) a global variable common to your ISR.
*/

void startsync(void);
void startmultisync(int iDeviceNum);
void endsync(void);
void endmultisync(int iDeviceNum);

#ifdef __cplusplus
}

#endif

#endif
```

LIST OF REFERENCES

[1] Enrico Buracchini, “The Software Radio Concept” *IEEE Communications Magazine,* pp. 138-143, September 2000.

[2] Joseph Mitola, “The Software Radio Architecture” *IEEE Communications Magazine*, pp. 26-38, May 1995.

[3] Proakis, J.G. and Manolakis, D.G., *Digital Signal Processing: Principles, Algorithms and Applications*, 3d edition, Prentice Hall, New Jersey, 1996.

[4] Reed, J.H., *Software Radio: A Modern Approach to Radio Engineering,* Prentice Hall, New Jersey, 2002.

[5] Mitola, J., *Software Radio Architecture: Object-Oriented Approaches to Wireless Systems Engineering,* John Wiley & Sons, New York, 2002.

[6] Shepherd G. and Krunglinski D., *Programming with Microsoft Visual C++ .NET*, Microsoft Press, Redmond, 2003.

[7] Red River Engineering, Document No. REF-303-000-R01, *WaveRunner Plus Channel Surfer Channel Blaster Hardware Reference Manual,* 2003.

[8] Intersil, Document FN6013.1, *ISL5216 Four-Channel Programmable Digital Down Converter Data Sheet,* February 2002.

[9] Intersil, Document FN6004.2, *ISL5217 Quad Programmable Up Converter Data Sheet,* March 2003.

[10] Capt. Nikolaos Apostolou, Hellenic Army, “Signal Synthesis With Dynamically-Changing Power Spectral Density, in a Software Defined Radio Transmitter”, Masters Thesis, September 2003, Naval Postgraduate School, Monterey, California.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Jovan Lebaric
   Naval Postgraduate School
   Monterey, CA

4. Professor Curtis Schleher
   Naval Postgraduate School
   Monterey, CA

5. Professor Roberto Cristi
   Naval Postgraduate School
   Monterey, CA

6. Jan E. Tighe, CDR USN
   Naval Information Warfare Activity
   Washington, DC

7. Georgios Zafeiropoulos
   53 Attalou Str. Kamatero
   Athens 13451
   Greece

THIS PAGE INTENTIONALLY LEFT BLANK